Experimental Study of Minimum Cut Algorithms

by

Matthew S. Levine

A.B. Computer Science
Princeton University, 1995

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MAY 1997

Signature of Author ....
Department of Electrical Engineering and Computer Science
May 9, 1997

Certified by ....
David R. Karger
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Experimental Study of Minimum Cut Algorithms

by

Matthew S. Levine

Submitted to the Department of Electrical Engineering and
Computer Science on May 9, 1997 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in Computer Science

Abstract

Recently, several new algorithms have been developed for the minimum cut problem that substantially improve worst-case time bounds for the problem. These algorithms are very different from the earlier ones and from each other. We conduct an experimental evaluation of the relative performance of these algorithms. In the process, we develop heuristics and data structures that substantially improve practical performance of the algorithms. We also develop problem families for testing minimum cut algorithms. Our work leads to a better understanding of practical performance of the minimum cut algorithms and produces very efficient codes for the problem.

Keywords: minimum cut, graph algorithms, experimental evaluation, network optimization

Thesis Supervisor: David Karger
Title: Assistant Professor of Electrical Engineering and Computer Science

This research was partly supported by DARPA contracts N000014-95-1-1246 and DABT63-95-C-0009, Army Contract DAAH04-95-1-0607 and NSF Award CCR-9624239. Some of this work was done at NEC Research Institute.

Contents

Chapter 1

Introduction

A *minimum cut* of an n-vertex, m-edge, capacitated, undirected graph is a partition of the vertices into two sets that minimizes the total capacity of edges with endpoints in different sets. This concept is more natural in pictures than in words; see Figure 1.1.

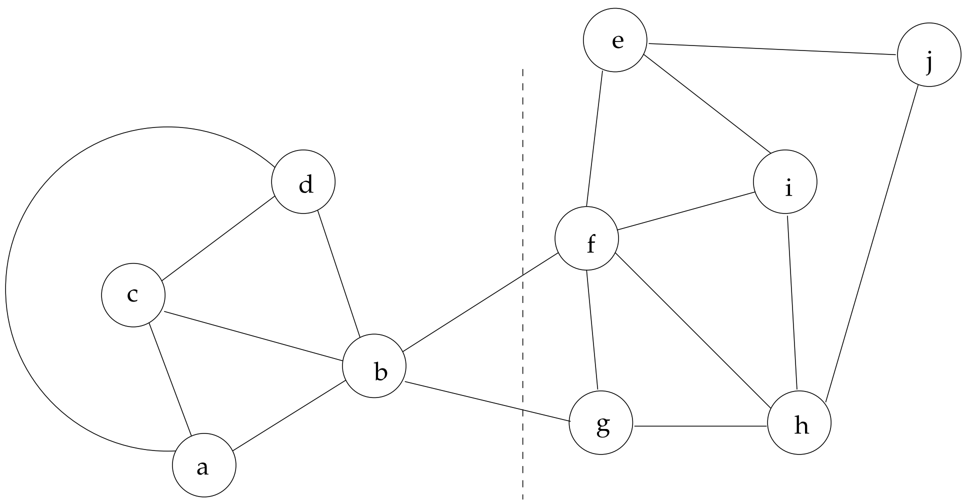

Figure 1.1: The dashed line shows a minimum cut of this graph.

Computation of minimum cuts is useful in various applications. An easy example is network reliability theory [37, 53]. If edges of a network fail with some probability, it makes intuitive sense that the greatest danger of network disconnection is at a minimum cut. Minimum cuts also arise in information retrieval [6], compilers for parallel languages [7], and cutting-plane algorithms for the Traveling Salesman Problem (TSP) [3]. We have also received requests for our codes from researchers interested in routing in ATM networks and computational biology.

1.1 Previous Work

The problem of finding a minimum cut has a long history. It was originally considered a harder variant of the minimum s-t cut problem, which places the further restriction that designated vertices s and t be on opposite sides of the partition. The well-known max-flow–min-cut theorem [22, 21] implies that an s-t minimum cut can be found by computing an s-t maximum flow. In 1961, Gomory and Hu showed how to solve the minimum cut problem with $n - 1$ s-t minimum cut computations. Subsequently there was much progress in computing maximum flows, but no one has yet been able to prove a time bound better than $O(nm)$ for any of the best algorithms [1, 9, 10, 25, 41]. Hence we cannot give a bound better than $O(n^2m)$ for the Gomory-Hu algorithm.

Gomory-Hu stood as the best algorithm for the problem until 1989, when Nagamochi and Ibaraki [47] showed how to find a minimum cut without using maximum flows. Their algorithm (which we will call NI) runs in $O(n(m+n\log n))$ time. In 1992, Hao and Orlin [29, 30] rejuvenated the flow approach by showing that clever modification of the Gomory-Hu algorithm implemented with a push-relabel maximum flow algorithm runs in time asymptotically equal to the time to compute one maximum flow: $O(nm\log(n^2/m))$. (We refer to this algorithm as HO.)

Progress continued in 1993 with a randomized algorithm (KS) given by Karger and Stein [39, 40] . With probability at least $1 - 1/n$, it finds all minimum cuts in $O(n^2\log^3 n)$ time. Finally, in 1996, Karger [38] gave two closely related algorithms (K). The first finds a minimum cut with probability at least $1 - 1/n$ and runs in $O(m\log^3 n)$ time; the second finds all minimum cuts with probability at least $1 - 1/n$ and runs in $O(n^2\log n)$ time.

The recent burst of theoretical progress has outdated implementation experiments. In 1990, Padberg and Rinaldi published a study on practical implementation of Gomory-Hu [51], which is very valuable for the design of heuristics, but unfortunately came just before the theory breakthroughs. Nagamochi *et al* [48] confirm that NI often beats Gomory-Hu in practice. Nothing is known about the practical performance of the other new algorithms.

1.2 Our Contribution

In this paper we address the question of the practical performance of minimum cut algorithms. We consider all of the contenders: NI, HO, KS, and K. Our goal is to obtain efficient implementations of all the algorithms and meaningful comparisons of their performance. Accomplishing this goal has two main aspects: obtaining good implementations and obtaining good tests.

A major aspect of obtaining good implementations is making good use of heuristics. We introduce a new strategy for applying the heuristics of Padberg and Rinaldi, which turns out to be very important to the efficiency of our implementations. We give a modified version of KS that seems to be more effective. We also introduce a new heuristic for maximum-flow based cut algorithms.

For both KS and K, which are Monte Carlo algorithms, guaranteeing correctness of our implementations required using parameters from the theoretical analysis. Thus we rework the analysis for both of these algorithms to get the best constant factors, and end up having to prove new theorems. In both cases it turns out that we believe there are stronger results than we can prove, so the practical performance of both of these algorithms stands to be improved directly by further theoretical work.

For K, the constants that we get from the theoretical analysis are unmanageably large, but we discovered that we always get the right answer with much smaller constants. Hence we cheat, and use a value that we cannot justify in our implementation. For this reason our implementation of K must be considered a heuristic—we do not have a proof of correctness.

For many applications, HO appears to be the best algorithm, followed by NI, but overall, our tests show that no single algorithm dominates the others. In general, HO and NI dominate K and KS, but on one problem family both K and KS show better asymptotic performance than NI and HO. We also have problem families where HO is asymptotically better than NI and vice versa.

Unfortunately, development and testing is iterative and interrelated. We develop tests that are hard for the implementations by looking for weaknesses in the implementations. Meanwhile, we use the tests to find weaknesses in the implementation and devise heuristics to improve performance. Thus it is difficult to be sure when we are done. For HO we take advantage of implementation work on maximum flow algorithms [2, 13, 16, 17, 50]; for NI we take advantage of the work of Nagamochi *et al*. KS and K were both developed from scratch, so it remains possible that their inferior performance is due to the fact that we are the first to develop heuristics for them.

Nevertheless, at the very least, we make significant progress in understanding how to implement these algorithms, introduce new heuristics, and give an interesting set of problem generators.

Note that this paper represents joint work with Chandra Chekuri, Andrew Goldberg, David Karger, and Clifford Stein. A preliminary version appeared in SODA 97 [8].

The paper is organized as follows. In Chapter 2 we review the theory behind the minimum cut algorithms, including definitions, characterizations of the problem, and descriptions of the algorithms. In Chapter 3 we discuss general implementation issues and details of each algorithm in turn. In Chapter 4 we discuss our experiments, including descriptions of the problem generators and results. Note that some readers will want to skip certain portions of this paper. In particular, readers who are already familiar with the algorithms may want to skip most of chapter two, and readers who are interested primarily in the bottom line may wish to skip all the way to the results section. We give warnings at specific places in the text before particularly complicated and/or detailed discussions that many readers will likely want to skip.

Chapter 2

Background

In this chapter we discuss the theory behind minimum cuts. One of the reasons minimum cuts are so interesting to study is that the theory behind the different algorithms is so varied. First we review approaches based on a reduction of the problem to the maximum flow problem. Next we look at algorithms that identify edges that cannot be in the minimum cut and use that information to reduce the problem. We conclude with algorithms based on packing trees, a dual problem.

We begin by introducing some terminology.

Let $G = (V, E, c)$ be an undirected graph with vertex set V, edge set E and non-negative real edge capacities $c : E \to \mathbb{R}^+$. Let $n = |V|$ and $m = |E|$. We will denote an undirected edge with endpoints v and w by $\{v, w\}$, but use $c(v, w)$ as shorthand for $c(\{v, w\})$. A *cut* is a partition of the vertices into two nonempty sets A and $\overline{A}$. The *capacity* or *value* of a cut $c(A, \overline{A})$ is defined by

$$c(A, \overline{A}) = \sum_{u \in A, v \in \overline{A}, \{u,v\} \in E} c(u, v) \tag{2.1}$$

We will sometimes unambiguously refer to a cut just by naming one side, and use the shorthand $c(A) = c(A, \overline{A})$. Also, if $A = \{v\}$, we may use $c(v)$ instead of $c(A)$. We refer to such cuts as *trivial*, and refer to the value $c(v)$ as the *capacity of* v. The edges included in the sum in (2.1) will be referred to as edges *in the cut* or edges *that cross the cut*. A *minimum cut* of G is a cut A that minimizes $c(A)$. We use $\lambda(G)$ to denote the value of the minimum cut.

Note that there may be more than one minimum cut. In fact, in a cycle where every edge has the same capacity, there are $\binom{n}{2}$ of them. (The cycle is actually the worst case. This result is shown by Dinitz, Karzanov, and Lomonosov [18] and also follows easily from the correctness proof of KS.) Since we only look for one minimum cut, we sometimes fix one minimum cut and refer to it as "the minimum cut".

2.1 Flow Based Approaches

The first approach to solving the minimum cut problem was based on a related problem, the minimum s-t cut problem. An s-t cut is a cut that has s and t on opposite sides of the partition.

The minimum s-t cut, $\lambda_{s,t}(G)$, is the s-t cut of minimum value. The well-known max-flow–min-cut theorem [22, 21] implies that a minimum s-t cut can be found by computing the maximum flow between s and t. In this section we discuss maximum flow and the minimum cut algorithms based on it.

2.1.1 Definitions

Although we study an undirected version of the minimum cut problem in this paper, flows are more natural in directed graphs. We transform an undirected graph into a directed graph in a standard way: replace each edge $\{v, w\}$ by two arcs (v, w) and (w, v), each with the same capacity. A cut in a directed graph is defined as in an undirected graph, except that we only count the edges in one direction. That is, the capacity of a cut A in a directed graph is the sum of the edges crossing from A to $\overline{A}$:

$$c(A, \overline{A}) = \sum_{u \in A, v \in \overline{A}, (u,v) \in E} c((u, v))$$

In general, $c(A, \overline{A})$ is not the same as $c(\overline{A}, A)$ in a directed graph, but since we replace each undirected edge with two directed edges, one in each direction, the value of every directed cut in the transformed graph is the same as the corresponding undirected cut in the original graph.

Let $G = (V, E, c)$ be a directed graph with two distinguished vertices s (source) and t (sink). A *flow* is a function $f : E \to \mathbb{R}$ satisfying

$$f(v, w) \leq c(v, w) \ , \quad \forall (v, w) \in E \tag{2.2}$$

$$f(v, w) = -f(w, v) \ , \quad \forall (v, w) \in E \tag{2.3}$$

$$\sum_{v \in V} f(v, w) = 0 \ , \quad \forall w \in V - \{s, t\}. \tag{2.4}$$

The first condition says that the flow on a directed edge is never more than the capacity of that edge. The second says that flow on an edge is antisymmetric: a units of flow on (u, v) implies $-a$ units of flow on (v, u). The final condition says that flow is conserved everywhere but the source and sink: the flow into each vertex is the same as the flow out of it.

We define the *residual capacity* $c_f(v, w)$ of an edge (v, w) to be $c_f(v, w) = c(v, w) - f(v, w)$. The *residual network* $G_f = (V, E_f)$ is the network induced by the edges that have non-zero (*i.e.* positive) residual capacity.

The *value* of a flow is the net flow into the sink, i.e.,

$$|f| = \sum_{v \in V} f(v, t).$$

It can be shown that the third condition, flow conservation, implies that this value is the same as the net flow out of the source.

The *maximum flow problem* is to determine a flow f for which $|f|$ is maximum. The max-flow–min-cut theorem states that the value of the maximum s-t flow is equal to the value of the minimum s-t cut, *i.e.*, $|f| = \lambda_{s,t}(G)$. Therefore all the edges of a minimum s-t cut are used up to capacity by a maximum s-t flow, and it can be shown that any s-t cut that is not minimum always has some edges with residual capacity. It follows that the vertices reachable from the source by edges in the residual network define an s-t minimum cut. An s-t maximum flow algorithm can thus be used to find an s-t minimum cut, and minimizing over all $\binom{n}{2}$ possible choices of s and t yields a minimum cut.

2.1.2 Push-Relabel Methods

As this paper is primarily concerned with minimum cuts, not maximum flows, we do not wish to get too involved with maximum flow algorithms. Conveniently, however, the fastest current maximum flow algorithms and the Hao-Orlin minimum cut algorithm are both based on the push relabel method, so we review that method here. For a more detailed description see [26].

We begin with some additional definitions. The algorithm maintains a *preflow*, which is a relaxed version of a flow. A preflow satisfies conditions (2.2) and (2.3), and the following relaxation of condition (2.4):

$$\sum_{v \in V} f(v, w) \geq 0 \ , \forall w \in V - \{s, t\} \tag{2.5}$$

So a preflow only has "one-sided" flow conservation. Flow still may not be created at a vertex, but now it may be absorbed, because we allow more flow to enter than leave. We define the *excess* at vertex v with respect to preflow f by

$$e_f(w) = \sum_{v \in V} f(v, w)$$

Given a preflow f, a *distance labeling* is a function $d : V \to \mathbb{N}$ that satisfies $d(v) \leq d(w) + 1$ for every (v, w) in the residual graph and $d(s) - d(t) \geq n$. The main point of this definition is that $d(v) - d(w)$ is always a lower bound on the distance from v to w in the residual graph. In general we hope for these lower bounds to be close to correct, so that we can use a distance labeling to direct flow to the sink along short paths, which at least intuitively is a good thing to do. Another way to phrase this intuition is that a distance labeling gives a "locally consistent" estimate on the distance to the sink. The idea is that if we maintained exact distances we would be able to route flow to the sink on a shortest path, but then we would have to do a lot of work to update the labels. By relaxing the conditions on labels so that they are only lower bounds on distances, we attempt to get the benefit of having distances without doing so much work. Since the labels give lower bounds on distances, $d(v) \geq d(t) + n$ implies that t is not reachable from v in G_f, because all paths have less than n edges. Thus the second condition, $d(s) - d(t) \geq n$, says that the sink is not reachable from the source, which means that some s-t cut is saturated, which means that if f is actually a flow then $|f|$ is maximum. We say that an arc $(v, w) \in E_f$ is *admissible* if $d(v) = d(w) + 1$. We say that a vertex v is *active* if the excess $e_f(v) > 0$.

Given a preflow f and a distance labeling d, we define *push* and *relabel* operations, which update f and d, respectively, as follows. The push operation applies to an admissible arc (v, w) where v is active; it increases flow on (v, w) by as much as possible: $\min(c_f(v, w), e_f(v))$. The relabel operation applies to an active vertex v with no outgoing admissible arcs. It sets $d(v)$ to the highest value allowed by the distance labeling constraints: one plus the smallest distance label of a vertex reachable from v via a residual arc. It is not hard to show that pushes and relabels preserve the validity of the distance labeling.

```
Push((v, w))
    (applies when (v, w) is a residual arc and d(v) = d(w) + 1)
    send min(c(v, w), excess(v)) units of flow along (v, w)
    remove (v, w) and/or add (w, v) to the residual graph if necessary
```

```
Relabel(v)
    (applies when v has excess, is not the sink, and for all residual (v, w) has d(v) ≠ d(w) + 1)
    d(v) = min_{residual (v,w)} d(w) + 1
```

The generic push-relabel algorithm for finding a minimum s-t cut starts by setting all distance labels to zero. Then the algorithm sets $d(s) = 2n - 1$ [1] and saturates all arcs out of s. This action gives the initial preflow and distance labeling. The algorithm applies push and relabel operations in an arbitrary order. When no operation applies, the algorithm terminates. Since one of push or relabel will always apply at an active vertex, termination means that there are no active vertices, which means that we have a flow. By the arguments above, this flow is maximum.

```
GenericPushRelabel(G, s, t)
    for all v, d(v) ← 0
    d(s) ← 2n − 1
    saturate all arcs out of s
    while there exists a vertex with excess
        find a place to apply a Push or Relabel and do so
    return excess at t
```

We get time bounds by counting the number of push and relabel operations. It is easy to show that distance labels only increase and are always $O(n)$, from which it follows that there can be only n^2 relabels. Each relabel requires looking at the outgoing edges of a vertex, so the total relabeling time is $\sum_v O(n)\mathit{degree}(v) = O(nm)$. We account for pushes by distinguishing pushes that *saturate*, that is, use all the residual capacity of a residual arc, and those that do not. After a saturating push on an arc, it is no longer part of the residual graph, and it cannot return to the residual graph until there is a push on the reverse arc. But for the reverse arc to be admissible, we must relabel one endpoint. Thus there can be only $O(n)$ saturating pushes per arc, giving a total of $O(nm)$ saturating pushes. It remains to bound the number of non-saturating pushes. It is possible to give a generic bound of $O(n^2m)$ on the number of non-saturating pushes, but we can get better bounds by considering variations on the algorithm.

At a high level, push-relabel algorithms differ by the order in which they apply push and relabel operations. One convenient way of ordering the operations is to define a *discharge* operation, which combines the push and relabel operations at a low level. The discharge operation applies to an active vertex v. The operation applies push operations to arcs out of v and relabel operations to v until v is no longer active.

[1] For an s-t cut computation, we can set $d(s) = n$; the higher value is needed for the Hao-Orlin algorithm.

```
Discharge(v)
    (applies when v has excess)
    while v has excess
        if v.currentArc = NIL, Relabel(v)
        else Push(v.currentArc)
```

We can now specify an ordering of pushes and relabels by giving a strategy for selecting the next active vertex to discharge. One possibility is the *highest label* strategy: discharge an active vertex with the highest distance label. This strategy, in combination with appropriate heuristics, seems to give the best results in practice [13]. It also permits a better bound on the number of non-saturating push operations: $O(n^2\sqrt{m})$ [11].

```
HighestLabelPushRelabel(G, s, t)
    for all v, d(v) ← 0
    d(s) ← 2n − 1
    saturate all arcs out of s
    while there exists an active vertex
        Discharge(an active vertex with maximum distance label)
```

We now sketch the proof of this time bound. Call the time between successive relabels a *phase*. At the end of a phase, for any vertex that has excess that was moved during the phase, we can identify a "trajectory" of non-saturating pushes that contributed to the excess. These trajectories end either at an edge that had a saturating push, or a vertex that had done no pushes since it was last relabeled. The key observation is that a non-saturating push from a vertex with the highest label makes that vertex inactive and, since there are no higher labeled vertices that could push to it, it must stay inactive at least until a relabel occurs. Thus these trajectories are vertex disjoint. So there can only be $\sqrt{m}$ trajectories longer than $n/\sqrt{m}$ in a phase, totaling $l\sqrt{m}$ non-saturating pushes for a phase in which the maximum distance label (over active vertices) drops by l. But the total increase in distance labels is only $O(n^2)$, so the total decrease is the same, giving a bound of $O(n^2\sqrt{m})$ on the number of non-saturating pushes in trajectories longer than $n/\sqrt{m}$. And since each trajectory ends an edge that had a saturating push or a vertex that was newly relabeled, the total number of trajectories is only $O(nm)$, so the total number of non-saturating pushes in trajectories shorter than $n/\sqrt{m}$ is also only $O(n^2\sqrt{m})$.

Another possibility is to use a FIFO queue to order discharge operations. In conjunction with dynamic trees, a sophisticated data-structure that makes it possible to do many non-saturating pushes at once, this method gives the best known time bound: $O(nm\log(n^2/m))$ [26]. We did not implement this version, and the full description is rather involved, so we do not give it here.

We assume that a relabel operation always uses the *gap relabeling heuristic* [12, 16]. This heuristic often speeds up push-relabel algorithms for the maximum flow problem [2, 13, 16, 50] and is essential for the analysis of the Hao-Orlin algorithm. Gap relabeling is based on the observation is that if there is no vertex with distance label x, then no excess at a vertex with distance label greater than x can reach the sink. (Consider applying discharge operations to these vertices before applying discharge operations to any of the other vertices. Since there is no vertex with distance label x, it will never be possible to push any excess to a vertex with label less than x. It follows that all this excess must return to the source.) We exploit this observation as follows. Just before relabeling

v, check if any other vertex has label $d(v)$. If the answer is yes, then relabel v. Otherwise, delete all vertices with distance label greater than $d(v)$. Note we have written the above as it applies to finding minimum cuts; if we actually want the flow then we cannot delete active vertices, but we can assign label $d(s)$ to all the vertices with label greater than $d(v)$, which is still helpful.

```
GapRelabel(v)
    (conditions for Relabel apply)
    if v is the only vertex with distance label d(v)
        remove all w with d(w) ≥ d(v)
    else Relabel(v)
```

2.1.3 The Gomory-Hu Algorithm

In 1961, Gomory and Hu [27] showed that $\lambda_{s,t}(G)$ for all $\binom{n}{2}$ pairs of s and t could actually be computed using only $n-1$ maximum flow computations. Their method immediately yields an algorithm for computing minimum cuts using only $O(n)$ maximum flow computations. Note that since Gomory and Hu considered directed graphs and actually compute all $\lambda_{s,t}(G)$, they solve a more general problem.

We can see more directly that $O(n)$ maximum flow computations suffice to compute a minimum cut. Fix some vertex s arbitrarily. In the minimum cut, there is some vertex t on the other side of the partition. For this t, the s-t minimum cut is clearly the same as the minimum cut. Therefore we can find the minimum cut by finding the minimum (over t) of minimum s-t cuts. This algorithm computes a minimum cut with $n-1$ minimum s-t cut computations. For the purposes of later discussion, we refer to this simplified algorithm as GH.

2.1.4 The Hao-Orlin Algorithm

A natural question to ask about GH is whether some of the information computed in one maximum flow computation can be reused in the next one. Hao and Orlin answer this question in the affirmative. The key new idea is to use a push-relabel maximum flow algorithm to implement GH, and use the preflow and distance labeling from the last max-flow computation as a starting point for the current one. This method allows us to amortize the work of the $(n-1)$ s-t cut computations to obtain a worst-case time bound that is asymptotically the same as the bound for one maximum flow computation. We give a brief description of this algorithm below. See [31] for details. Note that the algorithm given by Hao and Orlin applies to directed graphs, as did the original Gomory-Hu algorithm. As with GH, we ignore those details in this discussion.

A key concept of the Hao-Orlin algorithm is that of a *sleeping layer* of vertices. A sleeping layer is a set of vertices that do not participate in the current flow computation; there can be multiple such layers. A vertex is *asleep* if it belongs to a sleeping layer and *awake* otherwise. Initially all vertices are awake. When gap relabeling discovers a set of vertices disconnected from the sink, these vertices form a new sleeping layer. This layer is deleted from the graph and put on a stack of layers. When a layer of vertices is put to sleep, the values of the vertex distance labels are the same as they were just before the relabeling operation during which the layer was discovered. At some point during the execution of the algorithm, the top layer will be popped from the stack and the vertices of this layer will become awake. The point of the sleeping layers is that at the time we find them they are not relevant to the current flow computation, but we have done work to get their distance labels to the current state, so we save this information for use in a later flow computation.

The Hao-Orlin algorithm starts as follows. We select the first source and sink arbitrarily. We set the distance label of the source to $2n-1$ and saturate all arcs out of the source. Distance labels of all other vertices are set to zero. Then we start the first s-t cut computation. After an s-t cut computation terminates, we examine the cut it finds and remember the cut if its capacity is smaller than that of the best cut we have seen so far. Then we start the next computation as follows. First we set the distance label of t to $2n-1$ and saturate all of its outgoing arcs. This effectively makes it part of the source, so we refer to such vertices as *source vertices*. Next we look for a new sink. If there are no non-source, awake vertices, we awaken the top sleeping layer. We now pick the non-source, awake vertex with the smallest distance label as the new sink. If we cannot find a new sink because there are no non-source, awake vertices and there are no more sleeping layers, then all vertices are source vertices and we are done.

```
HO(G)
    λ̂ ← ∞
    designate some vertex s, give it label 2n − 1, and saturate all of its outgoing arcs
    while there are non-source vertices
        if there are no awake vertices, awaken the top sleeping layer
        pick the awake vertex with minimum distance label as t
        PushRelabel(G, s, t) (always using GapRelabel, not Relabel)
        if the excess at t is less than λ̂
            λ̂ ← excess at t
        designate t a source vertex, and saturate all of its outgoing edges
    return λ̂
```

It is not hard to check that the distance labels remain valid throughout the computation, which implies the correctness of the algorithm. Likewise, as in the maximum flow context, the distance labels are $O(n)$ and only increase. It follows that using highest label selection, the time bound for HO is $O(n^2\sqrt{m})$. The proof for FIFO selection with dynamic trees also carries over, giving a time bound of $O(nm\log(n^2/m))$.

2.2 Contraction

Another way to approach the minimum cut problem is to try to identify vertices that are on the same side of the minimum cut. Given two such vertices, we would like to reduce the problem. This motivates the following definition:

Given a graph G and vertices v and w, we create $G/\{v,w\}$, the *contraction* of v and w, by merging v and w into one node. That is, v and w cease to be discernible vertices; there is only a node representing the two of them, which has as its neighbors the union of the neighbors of v and the neighbors of w. If $\{v,w\} \in E$, we often refer to the contraction of v and w as the contraction of edge

$\{v, w\}$. Multiple edges are preserved by this operation, at least in terms of capacity. That is, if v and w have a common neighbor u, then in $G/\{v, w\}$ either $\{vw, u\}$ has capacity $c(v, u) + c(w, u)$ or there are two edges $\{vw, u\}$, one with capacity $c(v, u)$ and the other with capacity $c(w, u)$. These two views are equivalent in theory; which paradigm to use when representing the graph in practice is an implementation detail.

Note that although the terms *node* and *vertex* are often used interchangeably, we make a distinction for the purposes of talking about contracted graphs. We use the term node for the base set of a graph potentially created by a contraction operation, and the term vertex for the input. So after any series of contractions, nodes correspond to sets of vertices of the input graph.

The key property that we want from the contraction operation is captured in the following two lemmas:

Lemma 2.2.1 *Given a network G and two nodes v and w, if v and w are on the same side of some minimum cut, then $\lambda(G) = \lambda(G/\{v, w\})$.*

Proof. The proof is immediate from the fact that no new cuts are created by contraction and that by assumption all the edges of some minimum cut are not contracted. ■

Lemma 2.2.2 *Given a network G and two nodes v and w, $\lambda(G) = \min\{\lambda(G/\{v, w\}), \lambda_{v,w}(G)\}$.*

Proof. If v and w are on the same side of some minimum cut, then $\lambda(G) = \lambda(G/\{v, w\})$ by Lemma 2.2.1. Otherwise v and w are on opposite sides of every minimum cut, so $\lambda(G) = \lambda_{v,w}(G)$ by definition. ■

So given two vertices on the same side of any minimum cut, contraction produces a smaller graph with the same minimum cut. Further, given a minimum v-w cut, we can find a minimum cut by taking the smaller of the v-w cut we have and the minimum cut of $G/\{v, w\}$. From these observations we immediately get a high level minimum cut algorithm:

```
GenericContractCut(G)
    λ̂ ← ∞
    while G has more than one node
        Either
            1. identify an edge {v, w} that is not in some minimum cut
            2. compute λ_{v,w}(G) for some v and w and set λ̂ = min{λ_{v,w}(G), λ̂}
        G ← G/{v, w}
    return λ̂
```

Since a contraction reduces the number of nodes by one, this algorithm requires $n-1$ iterations of the while loop.

We assume that our algorithms always keep track of the minimum cut seen so far, as in GenericContractCut, so we refer to an edge $\{v, w\}$ as *contractible* if it is not in some minimum cut or if we already know $\lambda_{v,w}(G)$.

Note that GH can be modified to fit in this framework, because a maximum flow computation identifies a contractible edge (always option 2). Thus we use $n-1$ flow computations. Likewise in HO, we can contract the source and the sink at the end of each flow computation. Actually, HO already does contractions implicitly by designation of source vertices, but for the purposes of adding heuristics it turns out to be desirable to think about doing the contractions explicitly.

In the remainder of this section we describe several other ways to identify contractible edges. First we discuss local tests for contractibility given by Padberg and Rinaldi. These do not always apply, so they do not result in a minimum cut algorithm, but they are excellent heuristics. Then we discuss an algorithm of Nagamochi and Ibaraki that identifies at least one contractible edge by a graph search. Finally we discuss an algorithm of Karger and Stein, which shows that "guessing" contractible edges is good enough for a high probability of success.

2.2.1 The Padberg-Rinaldi Heuristics

In their implementation study of minimum cut algorithms, [51], Padberg and Rinaldi introduced several local tests for identifying contractible edges. The point is to try to take option 1 of GenericContractCut whenever possible. Used in GH, every time a test finds a contractible edge, we save one maximum flow computation. If the tests do not identify any contractible edges, then we have no choice but to use the flow computation. Since maximum flow computations are expensive, fast tests for contractibility are a big win—even if sometimes they do not apply—because we do not lose much if they fail and we gain a lot if they pass.

In their paper, Padberg and Rinaldi give a very general class of tests. Some of these would be quite time consuming, and in fact, could dominate the running time of the new minimum cut algorithms. We single out the four cheapest tests, which are reasonable to use. We refer to these as PR tests or PR heuristics. We say that a test *passes* if one of the conditions is satisfied, which implies that the edge is contractible. Recall for the following formulas that $c(v)$ denotes the total capacity incident to vertex v.

Lemma 2.2.3 *[51] Let $\hat{\lambda}$ be an upper bound on $\lambda(G)$. If $v, w \in V$ satisfy any of the following conditions:*

PR1 $c(v,w) \geq \hat{\lambda}$

PR2 $c(v) \leq 2c(v,w)$

PR3 $\exists u$ *such that* $c(v) \leq 2(c(v,w) + c(v,u))$ *and* $c(w) \leq 2(c(v,w) + c(w,u))$,

PR4 $c(v,w) + \sum_u \min(c(v,u), c(w,u)) \geq \hat{\lambda}$

then one of the following conditions must hold:

1. *v and w are on the same side of some minimum cut.*
2. *$\{v\}$ is a minimum cut.*
3. *$\{w\}$ is a minimum cut.*
4. *There is only one minimum cut and $\{v, w\}$ is the only edge that crosses it.*

5. *There is only one minimum cut and the edges whose capacities are included in the sum for test* PR4 *are the only edges that cross it.*

Written mathematically, these tests are difficult to interpret, but they are actually fairly intuitive. This intuition comes out best in the proof, so we give that now.

Proof. PR1 says that if we have an edge with capacity greater than an upper bound on the minimum cut value, then it is not in some minimum cut. This result is immediate from the fact that the value of any cut including $\{v, w\}$ is at least $c(v, w)$. If the capacity and bound are equal, then a minimum cut that includes $\{v, w\}$ can have no other edges. Thus if there is another minimum cut it cannot include $\{v, w\}$, and otherwise we have that condition 4 holds.

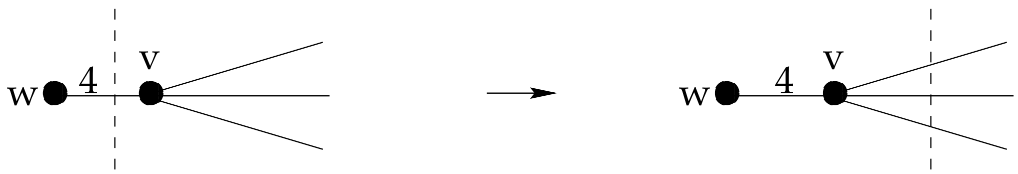

Figure 2.1: PR2: if $c(v, w) \geq c(v)/2$, then the cut on the right is no bigger than the cut on the left.

PR2 says that if we have an edge $\{v, w\}$ with capacity at least half the capacity of v, then either it is not in some minimum cut or $\{v\}$ is a minimum cut. To see this, consider such an edge (see Figure 2.1). If it does not cross some minimum cut then condition 1 holds, so suppose it crosses every minimum cut. Fix one of the minimum cuts. What happens if we move v to the other side of the vertex partition? The cut value loses $c(v, w)$ and may gain as much as $c(v) - c(v, w)$. But the second quantity is at most $c(v, w)$, so we cannot make the cut value larger, which contradicts the assumption that $\{v, w\}$ crosses every minimum cut. Of course, if $\{v\}$ is the minimum cut, then we cannot move v across the partition, but that is the only other possibility. Note that by symmetry, PR2 also passes if $c(w) \leq c(v, w)$.

PR3 is a more complicated form of PR2. Again, consider an edge where the conditions hold. Suppose u is on the same side of some minimum cut as v. Then by Lemma 2.2.1 we can consider $G/\{u, v\}$, because it has the same cut value. This merges $\{u, w\}$ and $\{v, w\}$, so the assumption that $c(w) \leq 2(c(v, w) + c(w, u))$ implies that PR2 applies at w in the contracted graph. Now suppose u is not on the same side of the minimum cut as v; that is, it is on the same side as w. By symmetry, the same argument applies. So either way, $\{v, w\}$ does not cross some minimum cut or one of $\{v\}$ or $\{w\}$ is a minimum cut. (See Figure 2.2)

PR4 is a generalization of PR1. We consider $\{v, w\}$ and all of the length two paths between v and w (see Figure 2.3). Any cut separating v and w must include $\{v, w\}$ and at least one edge from each of the paths of length two. The test is computing the minimum value that this quantity can have; clearly if it is greater than an upper bound on the cut value, then v and w cannot be on the same side of any minimum cut. And as in PR1, if the test is met with equality it is possible that the edges we have summed over are the only edges of the unique minimum cut, which means that case 5 holds. Note that any edge that passes PR1 will pass PR4, but we distinguish the two tests because PR1 is cheaper to compute. ∎

We still have not argued that these tests can help us. Observe first that conditions 4 and 5 are

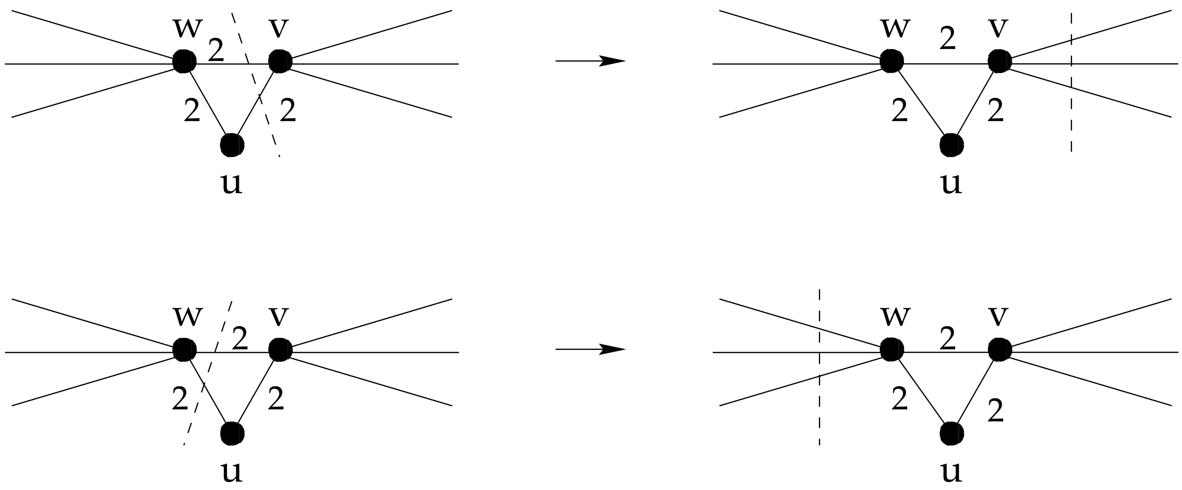

Figure 2.2: PR3: if $c(v,w)+c(v,u) \geq c(v)/2$ and $c(v,w)+c(w,u) \geq c(w)/2$, then for either case of a cut containing $\{v,w\}$ (on the left), there is a cut that does not contain $\{v,w\}$ and is no bigger (on the right).

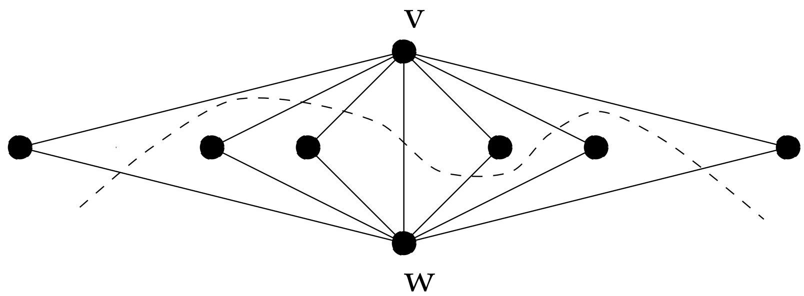

Figure 2.3: PR4: any cut separating v and w must also cut one edge from each of the paths of length two.

technicalities that will not concern us. In order to have $\hat{\lambda}$ set to $\lambda(G)$ we must have already found a minimum cut, so there is no question of missing the unique minimum cut by ignoring possibilities 4 and 5. Thus when $\{v, w\}$ passes a PR test, it either is not in some minimum cut, or one of $\{v\}$ or $\{w\}$ is a minimum cut. So as long as we check $c(v)$ and $c(w)$ and update $\hat{\lambda}$ if appropriate, $\{v, w\}$ is contractible if it passes a PR test.

Note that we need to be more careful if we want to find all minimum cuts. In that case we would want condition 1 strengthened to a guarantee that $\{v, w\}$ is not in any minimum cut. It is easy to show that we can get that condition by making the inequalities in the tests strict. Likewise, if we are interested in finding near-minimum cuts, we must relax the tests further. In particular, if we want to find all cuts with value at most $\alpha\lambda(G)$, we must introduce α into the inequalities of the tests. This detail is only an issue for KS and K, which are capable of finding all minimum cuts in the same time bounds, and can be extended to find near-minimum cuts.

It is not hard to see that it is possible to have a graph where none of these tests apply. Consider an uncapacitated graph with minimum cut at least 2 (PR1 fails), where each vertex has degree at least 3 (PR2 fails), and there are no triangles (PR3 and PR4 fail). An example is a cycle on $n - 2$ nodes with an extra vertex connected to every other node on the cycle and another extra vertex connected to the remaining nodes (see Figure 2.4). We call this graph a bicycle wheel (because of the heavy rim and light, interleaved spokes), and in fact test our codes on it.

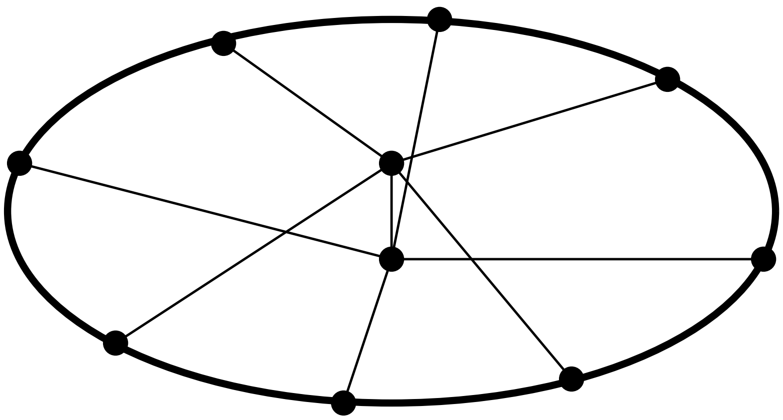

Figure 2.4: A graph on which all PR tests fail.

2.2.2 The Nagamochi-Ibaraki Algorithm

Other than by computing maximum flows, we still have not given a guaranteed way to identify a contractible edge. As discussed above, given a subroutine to find a contractible edge, we get a minimum cut algorithm: repeatedly find a contractible edge and contract it until the graph has only two vertices. This method requires $n - 2$ calls to the subroutine. So the question is whether there is a fast such subroutine. Nagamochi and Ibaraki [47] show that there is a surprisingly simple one.

For the purposes of intuition, consider uncapacitated multigraphs. Suppose we have a graph with minimum cut value one. Then the graph is connected, so we can find a spanning tree. For every minimum cut, the one edge of the cut must be in the tree, or else it is not spanning. Thus all the minimum cut edges are included in this spanning tree. Any other edges are not in any minimum cut, and are therefore contractible. Generalizing this idea, we define a *sparse* k*-connectivity certificate* to be a subgraph with at most kn edges in which the value of any cut is the minimum of k and the cut's original value. So for $k \geq \lambda$, if we find a sparse k-connectivity certificate, any edge not in the certificate is contractible. Further, as suggested by the example above, we can find such a subgraph by repeatedly (k times) finding a maximal spanning forest and removing those edges from further consideration.

There are still two problems with this idea. First, in a capacitated graph, it is not efficient to repeat any step λ times. Second, it is not guaranteed that we find a contractible edge, as the certificate may contain all of the original edges.

Nagamochi and Ibaraki solve these two problems simultaneously. They use a graph search called *scan-first search* that finds all of the maximal spanning forests in one pass over the graph, and they show that this search also finds a minimum v-w cut for some v and w, which guarantees that we will be able to do at least one contraction. Note that it is possible to invert this perspective. We can say that Nagamochi and Ibaraki gave an algorithm that efficiently finds a minimum v-w cut and also happens to find a sparse connectivity certificate, which provides a good heuristic for obtaining more contractions. It is a mistake, however, to proceed to ignore this heuristic, as it sometimes allows NI to finish in one search, instead of $n-1$.

In more detail, we build the maximal spanning forests by visiting the vertices in some order. When we visit a vertex, we assign to appropriate spanning forests all unassigned incident edges. In order to assign edges to forests, we keep track for each vertex of the maximum tree to which any incident has been assigned ($r(v)$). Let E_i be the edges of the ith maximal spanning forest. The remarkable theorem proved by Nagamochi and Ibaraki is that if we always pick the vertex with maximum $r(v)$ to visit next, and assign incident edge $\{v, w\}$ to $E_{r(w)+1}$, then we get the desired spanning forests.

```
ScanFirstSearch(G)
    for each v ∈ V
        r(v) ← 0
        mark v unvisited
    for each e ∈ E
        mark e unassigned
    E_1 ← E_2 ← ··· ← E_|E| ← ∅
    while there is an unvisited node
        v ← the unvisited node with largest r(v)
        for each unassigned {v, w}
            E_r(w)+1 ← E_r(w)+1 ∪ {v, w}
            if (r(v) = r(w)) r(v) ← r(v) + 1
            r(w) ← r(w) + 1
            Mark {v, w} assigned
        Mark v visited
```

We now sketch the proof that scan-first search finds the desired forests. First, note that we do indeed maintain the property that $r(w)$ is the maximum forest in which w has an incident edge. It follows that by adding $\{v, w\}$ to $E_{r(w)+1}$, we can never create a cycle, so we do indeed find forests. It remains to argue that when we add $\{v, w\}$ to forest $i+1$, there is a path from v to w in forest i. (If this statement is true, we are always adding each edge to the first forest in which it does not create a cycle, which implies that none of the previous forests could have been made larger by adding this edge, which means that we are properly simulating the idea of repeatedly finding maximal spanning forests and deleting them.) When assigning an edge to a tree, consider also directing it from the vertex being visited to the unvisited vertex. From the way we assign edges to forests, it is clear that each vertex has in-degree at most one in a given forest. Further, if we have two distinct trees in a given forest, each with at least one edge, then they cannot become connected, because the roots have already been visited, so there are no more unassigned incident edges to add there. Now, when we add $\{v, w\}$ to forest $i + 1$, we have $r(v)$ and $r(w)$ each at least i, so both of v and w have an incident edge in forest i. Assume for contradiction that there is no path between v and w in forest i, in which case they are in distinct trees. Let $v_0(w_0)$ be the root of the tree containing $v(w)$, and assume without loss of generality that v_0 is visited before w_0. Let $v_0, v_1, \ldots, v_h = v$ be the unique path from v_0 to v in the tree containing v. After visiting v_0, $r(v_1)$ is at least i, and $r(w_0)$ is at most $i-1$, since we assume w_0 is visited second. So we now visit v_1 before w_0, because it has a larger r. Repeating this argument, it follows that we visit v before we visit w_0, but in order for w_0 to be the root of the tree containing w, it must be visited before v, so we have a contradiction.

We now direct our attention to the second claim, that scan-first search also finds a minimum v-w cut for some v and w. Consider the last edge assigned. This edge necessarily connects the next-to-last vertex visited (v) to the last vertex visited (w). Each edge of w is clearly added to a different forest, so the $\{v, w\}$ edge is assigned to $E_{c(w)}$. But this means that any cut containing $\{v, w\}$ has value at least $c(w)$, or else we could have put the edge in an earlier forest. In particular, this means that $\lambda_{v,w} \geq c(w)$. Since $\lambda_{v,w}$ cannot be more than $c(w)$, we have that in fact $\lambda_{v,w} = c(w)$.

Thus we always compute $\lambda_{v,w}(G)$ for some v and w and can therefore always take option 2 of GenericContractCut. Note that unlike GH, NI does not pick v and w, it just finds the minimum v-w cut for some v and w. Further, it is possible that we find many edges to which we can apply option 1 of GenericContractCut.

Using an appropriate priority queue to pick the next vertex to scan, the search runs in $O(m + n \log n)$ time; thus the total time of NI is $O(n(m + n \log n))$. Note that we have not talked about capacitated graphs; but we do not need to explicitly construct the E_i, so for integer weights there is no problem. In fact the proofs carry over for real weights as well. See Section 3.4 for discussion of what is actually implemented.

Matula's $2 + \epsilon$ Approximation Algorithm

Recall that NI only guarantees a reduction of one node per search. This situation arises because we do not want to make a mistake. However, if we are willing to settle for an answer that is within a constant factor of the minimum cut, we can guarantee that the number of edges is reduced by a constant factor with each search. This result is due to Matula [46]. The algorithm is as follows:

```
MatulaApprox(G, ε)
    compute d, the minimum over v of c(v)
    compute a sparse (d/(2+ε))-connectivity certificate and contract all of the non-certificate edges
    recurse
    return the smaller of d and the result of the recursive call
```

This algorithm gives an answer within $2+\epsilon$ of the minimum cut. The argument that this result holds is easy. If the minimum cut is less than $d/(2+\epsilon)$, then the sparse certificate contains all of the minimum cut edges, so the graph given to the recursive call will have the same minimum cut. When the minimum cut is more than $d/(2+\epsilon)$, since the minimum cut is at most d, d is a $2+\epsilon$ approximation.

We now argue the running time, assuming that the total edge capacity is bounded by a polynomial in n; the results can be extended to arbitrary capacities [35]. Since the original graph had $dn/2$ edges, and the certificate has only $dn/(2+\epsilon)$, an $\Omega(\epsilon)$ fraction of the edges must be eliminated at each step. It follows immediately that the running time is $O(m/\epsilon)$ for uncapacitated graphs, or $O(m(\log n)/\epsilon)$ for capacitated graphs with total edge weight polynomial in n.

2.2.3 The Karger-Stein Algorithm

Karger and Stein take a very different approach to pick edges to contract. Rather than finding a contractible edge, they just pick an edge at random and contract it. The intuition behind this surprising action is that relatively few edges cross any given minimum cut (that is what makes the cut minimum), so there is a reasonable chance that the edge is in fact contractible.

In order to explain this algorithm, we start by describing a simpler algorithm of Karger [33], which shows one of the key ideas. Consider our capacitated graph as an uncapacitated graph with multiple edges to represent capacitated edges. Pick an edge at random. Clearly the probability it is in the minimum cut is only λ/m. Further, since each edge has unit capacity, each node must have at least λ incident edges, so $m \geq \lambda n/2$. (If v had fewer incident edges, $\{v\}$ would be a smaller cut.) So the probability of picking a minimum cut edge is at most $\frac{\lambda}{\lambda n/2} = 2/n$. We say a minimum cut *survives* a contraction if no edge that crosses it is contracted. It follows that the probability that a given minimum cut survives k successive contractions (being *contracted down* to $n-k$ nodes) is at least

$$(1-\frac{2}{n})(1-\frac{2}{n-1})\cdots(1-\frac{2}{n-k+1}) = \frac{n-2}{n}\frac{n-3}{n-1}\cdots\frac{n-k-1}{n-k+1} = \frac{\binom{n-k}{2}}{\binom{n}{2}} \qquad (2.6)$$

In particular, if we repeatedly contract random edges until the graph has two nodes, the minimum survives with probability at least $1/\binom{n}{2}$. It follows that we can repeat this algorithm $O(n^2 \log n)$ times to find a minimum cut with probability at least $1-1/n$. This algorithm also works for capacitated graphs; the only modification we need to make is that edges should be picked for contraction with probability proportional to capacity.

Unfortunately, the above algorithm does not run very fast. One iteration of a sequence of $n-1$ contractions can take $O(n^2)$ time; $O(n^2 \log n)$ iterations can take $O(n^4 \log n)$ time. However, Karger and Stein [39] point out that the highest probability of failure is when the graph is small.

In fact, if we contract down to $n/\sqrt{2}$ nodes, Equation 2.6 says that the probability the minimum cut survives is at least one half. Thus instead of contracting down to two nodes, it makes sense to use a recursive approach:

```
RecursiveContract(G)
    if G has less than 7 vertices, compute the minimum cut by brute force and return it
    repeat twice:
        contract down to ⌈1 + n/√2⌉ nodes, giving G′
        RecursiveContract(G′)
    return the minimum of the two answers from 2b
```

Note that the base case is not $n = 2$ for technical reasons: when n is less than 7, $\lceil 1 + n/\sqrt{2} \rceil$ is bigger than n.

Theorem 2.2.4 *[39] The recursive contraction algorithm runs in $O(n^2 \log n)$ time and finds the minimum cut with probability $\Omega(1/\log n)$.*

It follows that $O(\log^2 n)$ repetitions run in $O(n^2 \log^3 n)$ time and find a minimum cut with probability at least $1 - 1/n$. Note that since the probability of success holds for any minimum cut, the algorithm actually finds all of them with high probability.

```
KS(G)
    repeat RecursiveContract(G) O(log² n) times and return the smallest cut seen.
```

Notice that the constant in the O, the number of iterations we must run, depends on the exact analysis of the success probability of the recursive contraction algorithm. This point will cause us some trouble in the next section, when we modify the algorithm.

A New Variant of KS

We note that KS makes the pessimistic assumption that the total edge capacity when n nodes remained was $n\lambda/2$. Under this assumption, contraction to less than $n/\sqrt{2}$ nodes might not preserve the minimum cut with probability at least $1/2$. Consider, however, the case of two cliques joined by a single edge. In this case, the original algorithm is being overly conservative in contracting to only $n/\sqrt{2}$ nodes. It could in fact contract to a far smaller graph while still preserving the minimum cut with reasonable probability.

We give a variant of the recursive contraction algorithm that has better behavior in this respect. Unfortunately, we have been unable to prove that it does not have worse behavior when the graph really does have only $n\lambda/2$ edges. In some sense it is not terrible that we cannot prove it is never worse, as we hope that our experiments would reveal such a problem. However, our experiments cannot be exhaustive, so it would be nice to know that there is not a bad graph we did not think of. Even worse, it turns out that a tiny change to this variant would cause it to have infinite expected running time, so we do need to at least show a polynomial time bound, even though we cannot show the $O(n^2 \log^3 n)$ time bound that we would like. Note that we also *must* carefully analyze the success probability, as that is the only way to guarantee the correctness of our implementation.

We now describe the variant. As in the PR tests, $\hat{\lambda}$ is an upper bound on the minimum cut.

```
NewRecursiveContract(G)
    compute min_v c(v), and update λ̂ if appropriate
    if G has only two nodes, return λ̂
    repeat twice:
        mark each edge {v, w} independently with probability 1 − 2^(−c(v,w)/λ̂)
        contract all marked edges, giving G′
        NewRecursiveContract(G′)
    return λ̂
```

There are two main differences to explain. First, we do not need to stop at 7 nodes, because we do not get stuck there any more. (Recall that the old algorithm got stuck there because it tried to contract to a specific size, and that size was not smaller than n when n was less than 7. Now we mark edges, so we freely contract down to one.) Notice also that we do not even bother to stop at 2, which seems the natural stopping point, because the first step updates $\hat{\lambda}$ to the minimum cut of a two node graph. The second change is the way we pick edges for contraction. To see why the new method makes sense, again consider the probability that the minimum cut survives the contractions. Let $c_1, c_2, \ldots c_j$ denote the capacities of the minimum cut edges. The probability the minimum cut survives is

$$2^{-c_1/\hat{\lambda}} 2^{-c_2/\hat{\lambda}} \cdots 2^{-c_j\hat{\lambda}} = 2^{-\sum c_i/\hat{\lambda}} = 2^{-\lambda/\hat{\lambda}}$$

Since $\hat{\lambda}$ is an upper bound on λ, this probability is always at least $1/2$.

So the new algorithm preserves the minimum cut with the same probability, but it may contract more edges. For example, given two cliques joined by a single edge, if $\hat{\lambda}$ is close to λ $(= 1)$, the new algorithm will contract many more edges, reducing the depth of the recursion. Note, however, that we must be careful, because if $\hat{\lambda}$ is very far from λ, the probability of contraction will be very small and the recursion depth and running time could get very large. Thus by this modification we hope to do more contractions when there are excess edges and reduce the recursion depth, but in the process we introduce the risk of not contracting enough edges and increasing the recursion depth. So we must be careful.

We resolve this problem by noticing that we have a convenient upper bound on λ in the form of the minimum degree of the input graph. If we use this upper bound, then the probability that any given vertex v is not involved in a contraction is at most $2^{-c(v)/\hat{\lambda}}$, which is at most $1/2$. Thus we expect at least half of the nodes to be involved in a contraction during each execution of step (b), which means that we expect the contracted graph to have only $3n/4$ nodes. Since we can do the $O(n)$ contractions in $O(m)$ time, if we actually reduced to $3n/4$ nodes each time then the recurrence for the running time would be

$$T(n) = O(n^2) + 2T(3n/4) = O(n^{\log_{4/3} 2}) \approx O(n^{2.4})$$

Unfortunately, we are not guaranteed such a reduction, so we cannot use this recurrence relation. Nevertheless, we conjecture that the new algorithm's performance is in fact equal to that of the old one, but this remains to be proved.

It is tempting to stop worrying about the running time now, because we are implementing the algorithm, so we are interested more in what it actually does than the best bound we can prove, but it turns out that we must be very careful. Recall that in the original algorithm, the minimum cut survived to depth k of the recursion with probability $\Omega(1/k)$. That was fine originally, since the depth was known to be finite, but now we must be careful. Suppose we made our base case one node, instead of two. Now we do not terminate until the minimum cut is contracted, so the probability we do not terminate at depth k is $\Omega(1/k)$. Hence the expected depth of the recursion is infinite!

We will now argue that when the base case is two nodes, the running time is in fact polynomial. From above, we can say that the probability a given node is not contracted away is at most $1/4$. Thus a Chernoff bound tells us that when k nodes remain, the probability that we do not contract away $(1-\epsilon)k/4$ of them is at most $e^{-\epsilon^2 k/8}$. So while $k = \Omega(\log n)$, with high probability we do get a constant factor reduction at each step, so with high probability the recursion only descends to depth $O(\log n)$ before the graph is down to $c \log n$ nodes.

After this point, since the graph has at least three nodes before termination, we have total edge weight at least $3\lambda/2$. Thus any time we do random contractions, the probability that all edges survive is at most $2^{-3/2}$. Suppose now that a sequence of $k \log n$ contractions does not bring the graph down to two nodes. We have then that in $k \log n$ independent trials, an event that happens with probability at most $2^{-3/2}$ occurred $(k-c)\log n$ times. That is, taking $\epsilon = 2\sqrt{2} - 1 - 2\sqrt{2}c/k$, we have that we are exceeding the expected the number of non-contractions by a factor of $1+\epsilon$. Applying Chernoff bounds, we get that the probability of this happening is at most

$$e^{\frac{-\epsilon^2 k \log n}{8\sqrt{2}}}$$

For k sufficiently larger than c (but still constant), ϵ is constant, so we get that the probability is $n^{-\Omega(k)}$. Since the number of paths only grows as 2^k, a union bound tells us that the expected depth is $O(\log n)$, which implies a polynomial running time.

It remains to show the success probability of the new algorithm, which we need to determine how many iterations to run. Unfortunately, we cannot afford to be sloppy here, as whatever constant we compute will be hard-coded into an implementation. Since the algorithm is Monte Carlo, we have no other way to ensure that an implementation succeeds with an appropriate probability. Therefore, we devote the rest of this section to a careful analysis of the new algorithm. Readers not interested in the details of the analysis can safely skip it.

We begin by reviewing the success probability analysis of the old algorithm. The main idea is to consider the tree defined by the recursion. (A node represents a call to RecursiveContract, and its children are the recursive calls.) We can use the tree to write a recurrence for the success probability. In particular, if we define $p(k)$ to be the probability that a node at height k succeeds in finding the minimum cut of the graph it is given, then $p(0) = 1$ and

$$p(k) \geq 1 - (1 - \frac{1}{2}p(k-1))^2 = p(k-1) - \frac{(p(k-1))^2}{4}$$

This recurrence follows from the algorithm; we succeed if in either trial the minimum cut survives the contractions and the recursive call is successful. Since we already argued that the minimum cut survives the contractions with probability at least $1/2$, and the probability of a recursive call succeeding from a height k node is $p(k-1)$, we get the desired recurrence. The base case comes from the fact that the algorithm uses a deterministic strategy when the graph is small enough.

To solve this recurrence, we use a change of variables. Let $q(k) = 4/p(k) - 1$. This substitution gives the recurrence

$$q(k) = q(k-1) + 1 + 1/q(k-1)$$

It is easy to verify by induction that

$$k + 3 \le q(k) \le k + H_{k+2} + 3/2$$

Assembling, we get that

$$p(k) \ge \frac{4}{k + H_{k+2} + 3/2}$$

Now, since we succeed in a height k tree if and only if the minimum cut survives at some depth k node, and $H_k \le 1 + \ln(k)$, we get the following lemma:

Lemma 2.2.5 *The probability that the minimum cut survives at some recursion node of depth d in the RCA is at least $\frac{4}{d+\ln(d+2)+5/2}$.*

As the depth of the recursion is roughly (*i.e.*, up to small additive factors) $\log_{\sqrt{2}} n = 2\log n$, it follows that the success probability is $\Omega(1/\log n)$.

We now consider the new algorithm. As we already argued, the new algorithm also preserves the minimum cut with probability at least $1/2$ before recursing, so Lemma 2.2.5 holds. Unfortunately, we no longer know the depth of the recursion. Whereas in the old algorithm we were guaranteed a reduction in the number of nodes from n to $\lceil 1 + n/\sqrt{2} \rceil$ in each recursive call, in the new algorithm we only *expect* to reduce the number of nodes by a factor of $3/4$. We will therefore have to do some additional work to determine the depth at which the minimum cut is actually found. We begin by assuming that the upper bound $\hat{\lambda}$ has been set to λ. At the end of this section, we justify our assumption.

Our analysis is based on a network reliability analysis from [37]. That paper considers a graph in which each edge fails with probability p, and determines the probability that the graph remains connected. This problem is related to our objective as follows. Our goal is to show that at a certain recursion depth, the recursion has been terminated, which means that our graph has been contracted to a single node. That is, we want the set of contracted edges to span (connect) all of G. Inverting this objective, we can consider deleting the set of edges that were not contracted, and require that deleting these edges not disconnect the graph.

Now consider a particular recursion node at depth d. The graph at this node is the outcome of a series of independent "contraction phases" in which each edge is contracted with probability $1 - 2^{-1/\lambda}$ (by our assumption that $\hat{\lambda} = \lambda$). That is, the probability of not being contracted is $2^{-1/\lambda}$. It follows that at depth d, the probability that any edge is not contracted is $2^{-d/\lambda}$. We now invert our perspective as in the previous paragraph. We ask whether deleting the uncontracted edges leaves us with a single component. In other words: we consider deleting every edge of G with probability $2^{-d/\lambda}$, and ask whether the remaining (contracted) edges connect G.

The following is proven in [45] (see also [37]), using the fact that among all graph with minimum cut λ, the graph most likely to become disconnected under random edge failures is a cycle:

Lemma 2.2.6 *Let G have n edges and minimum cut λ. Then the probability that G is not connected after edge failures with probability $1-p$ is at most n^2p^λ.*

We might hope to apply this lemma as follows.

Corollary 2.2.7 *At a node at recursion depth $k \log n$, for $k > 2$, the probability that G has not been contracted to a single node is at most n^{2-k}.*

Proof. At depth $k \log n$, the (cumulative) probability of non-contraction for a given edge is $p = 2^{-(k \log n)/\lambda} = n^{-k/\lambda}$. Plugging into Lemma 2.2.6, we find that the probability we have not contracted to a single node is at most n^{2-k}. ∎

Unfortunately, this lemma is not sufficient to prove what we want. At depth $3 \log n$ in the recursion tree, there are n^3 recursion nodes. Although each one has only a $1/n$ chance of not being a leaf of the recursion tree, there is a reasonable chance that not all are leaves. We must therefore perform a more careful analysis.

We evaluate the probability of success as the product of two quantities: the probability that the minimum cut survives contraction to the given depth and the probability that the minimum cut is found by our algorithm given that it survives. Conditioning on the survival of the minimum cut makes our analysis somewhat complicated.

Given the conditioning event, there is some node N at depth $k \log n$ in which the minimum cut has survived. We would like to claim that at this point the contracted graph has only two nodes. Unfortunately, conditioning on the survival of the minimum cut means that no minimum cut edge has been contracted, a condition that breaks our reliability model.

To deal with this problem, we rely on the fact that our new algorithm examines the degrees of the nodes in its inputs. It therefore suffices to show that at least one side of the minimum cut is contracted to a single node, since this single node will be examined by the algorithm. We will in fact argue that both sides will be contracted to a single node. Another way to say this is that the edge failures break G into exactly two connected components.

Lemma 2.2.8 *Conditioned on the fact that a minimum cut has failed, the cycle is the most likely graph to partition into more than two pieces under random edge failures.*

Proof. A straightforward modification of [45]. ∎

Corollary 2.2.9 *Conditioned on the fact that a minimum cut has failed, the probability a graph partitions into 3 or more pieces is at most $np^{\lambda/2}$.*

The following lemma is an immediate corollary.

Lemma 2.2.10 *Conditioned on that fact that the minimum cut survives at some node of depth $k \log n$, the recursive contraction algorithm finds the minimum cut with probability at least*

$$f(k, n) = 1 - n^{1-k/2}.$$

Proof. Consider the depth $k \log n$ recursion node at which the minimum cut survives. At this depth, the probability of edge "failure" is $n^{-k/\lambda}$. From the previous lemma, $f(k, n)$ is the probability that the contracted edges at this recursion node reduce the graph to two nodes, implying we find the minimum cut. ∎

Lemma 2.2.11 *For any* k, *at depth* $k \log n$, *the new RCA finds the minimum cut with probability at least*

$$\frac{4}{k \log n + \ln(k \log n + 2) + 5/2} f(k, n).$$

Proof. From Lemma 2.2.5 we find that the probability that the minimum cut survives in some recursion node at depth $d = k \log n$ is at least $\frac{4}{d+\ln(d+2)+5/2}$. We now condition on the event having taken place and apply Lemma 2.2.10 to find the probability of success $f(k, n)$ given this event. The overall probability is the product of these two quantities. ∎

Corollary 2.2.12 *The probability that a single iteration finds the minimum cut is at least*

$$\frac{4}{2 \log n + 2 \log \log n + \ln(2 \log n + 2 \log \log n + 2) + 5/2}(1 - \frac{1}{\log n})$$

Proof. Set $k = 2 + \frac{2 \log \log n}{\log n}$ in Lemma 2.2.11 ∎

Note it is easy to evaluate the quantity of Lemma 2.2.11 on-line, so it is not necessary to analytically determine the optimal k. We give Corollary 2.2.12 just to show that the success probability is again roughly $\Omega(1/\log n)$.

From this analysis of the success probability of a single iteration, it is easy to compute the number of iterations needed to achieve a specified success probability. In particular, if we want success probability p, and we denote the maximum value (over k) of the probability in Lemma 2.2.11 as s, then i, the number of iterations we need is given by

$$(1 - s)^i < 1 - p$$

Finally we consider what happens if $\hat{\lambda}$ is more than λ. In this case, the probability of contracting a minimum cut edge at any recursion node is strictly less than one half. The result is that the probability of minimum cut survival at depth k quickly converges to a constant, instead of falling off linearly with k. (Note that we would get a similarly dramatic change if we made the probability strictly greater than one half: the probability would fall off exponentially with k.) In particular, for $\hat{\lambda} > \lambda$,

$$\Pr[\text{success}] > 2^{1+\frac{\lambda}{\hat{\lambda}}} - 2^{2\frac{\lambda}{\hat{\lambda}}}$$

Further, revising Lemma 2.2.10, we see that $f(k, n) = 1 - n^{1-\frac{k\lambda}{2\hat{\lambda}}}$. But since the probability of the minimum cut surviving at a given depth is at least a constant, we can consider going to an arbitrary depth, in which case this quantity becomes 1. Note that we could attempt to deliberately use $\hat{\lambda} > \lambda$, which would raise our success probability, but we would have to very careful, as we could easily cause the algorithm to run forever. This idea should be studied further.

2.3 Tree Packings

Recall that in the first section we approach the minimum cut problem by exploiting the nice duality between s-t minimum cuts and maximum flow. Given that this duality exists, it is natural to ask whether the minimum cut problem has its own dual, which we could exploit directly. It turns out that there is such a dual, and in this section we discuss algorithms that use it.

An a-*arborescence* is a directed spanning tree rooted at a; that is, a directed acyclic subgraph where a has in-degree zero and every other node has in-degree one. An a-*cut* is a cut; its value is the total capacity of edges crossing the partition from the side that includes a to the other. Two theorems of Edmonds relate a-cuts and a-arborescences:

Theorem 2.3.1 *[20] In a directed graph the maximum number of edge-disjoint a-arborescences equals the minimum value of an a-cut.*

Theorem 2.3.2 *[19] The edges of a directed graph can be partitioned into k a-arborescences if and only if they can be partitioned into k spanning trees where every vertex except a has in-degree k.*

We refer to a set of edge-disjoint trees as a *tree packing*. It follows that if we take our undirected graph and transform it into a directed graph as we did for maximum flows, then for an arbitrary a, the maximum cardinality of a tree packing where every node except a has the same in-degree is equal to the value of the minimum cut.

If we try to consider undirected spanning trees, we get a theorem that is close, but has some slack in it. In particular, Nash-Williams shows:

Theorem 2.3.3 *[49] An undirected graph with minimum cut λ contains at least $\lfloor \lambda/2 \rfloor$ edge-disjoint spanning trees.*

Note that NI packs spanning trees (and forests), but it does so with different intent, as it does not attempt to find a maximum packing, but rather a maximal one. NI also packs undirected spanning trees, which means that in general it cannot hope to find more than $\lambda/2$ full trees.

In this section we describe algorithms that use tree packings to find the minimum cut. First we review an algorithm of Gabow that runs in time proportional to the value of the minimum cut. We then give a strongly polynomial algorithm due to Karger. It is not necessary to understand Gabow's algorithm to understand Karger's, although it is used in the implementation. Some readers may wish to skip directly to the section on Karger's algorithm.

2.3.1 Gabow's Algorithm

Edmonds' theorems about the relation between arborescences and minimum cuts are analogous to the max-flow–min-cut theorem. It is therefore natural to look for an "augmenting trees" algorithm to find a tree packing, analogous to the classical Ford-Fulkerson augmenting paths algorithm to find a maximum flow [22]. This is precisely what Gabow [23] gives.

The augmenting paths algorithm for maximum flow works by repeatedly finding a path from the source to the sink in the residual graph and sending as much flow along it as possible. This does not mean that it is possible to greedily choose flow paths; a new augmenting path may eliminate the flow on some edge of a previous path, but this just amounts to the two paths exchanging segments. (If one path goes from s to v to w to t, and another goes from s to w to v to t, this amounts to one path going from s to v to t and the other going from s to w to t.)

Likewise, the basic idea of Gabow's algorithm is to repeatedly find a tree and delete it from the graph until it is impossible to find any more. Again, this is not to say that a greedy strategy works—on the contrary, in its attempt to find a new tree, the algorithm must consider changing the trees already found.

More precisely, given a set of trees, we try to build a new tree by starting with an empty forest (just vertices) and connecting trees of the forest until we have a new tree. To avoid confusion, we refer to a tree of the forest as an *f-tree*. We always hope to be able to find an edge that connects two f-trees, but it is possible that we cannot do so with unused edges. Nevertheless, we may be able to "trade" edges with an existing tree, taking an edge we need to link up two f-trees and giving an unused edge so that the tree stays connected. This process can of course be more complicated; we may need to move edges around many of the existing trees in order to get an edge we want.

Remember that there is also another restriction: we must ensure that if the maximum packing has k trees, then every node except a will have in-degree k. Notice that this restriction is over the entire packing. That is, we do not require that each node other than a have in-degree one in each tree, merely that the sum of a node's in-degrees over the trees is k. To handle this restriction, we maintain the invariant that each f-tree has precisely one node that does not have enough incoming edges (except for the tree that contains a, since a never needs incoming edges). We call such a node the *root* of its f-tree. (We always call a a root.) When we look for an edge that connects two f-trees, we actually want an edge that is directed to the root of one of them (never a). Then when we merge the two f-trees there is again precisely one node that needs incoming edges, and when we succeed in building a whole tree, a is necessarily the root, which implies that we have satisfied the degree constraints.

Given this framework, in order to make progress we need to find a way to both increase the in-degree of an f-tree root and connect that f-tree to another f-tree. Consider an f-tree root v (not a). The easy case is if v has an incoming edge from a vertex in another f-tree. Then we are all set. However, it is possible that all of v's incoming edges are from elsewhere in the same f-tree. In this case we need to try trading edges with some tree T. We give T one of v's incoming edges, e, so that v's in-degree will increase by one; T gives e', one of the edges on the cycle formed by adding e, back to the set of unused edges so that it will stay a tree. This exchange effectively changes the root of the f-tree to v', the head of e', because that is the node that now lacks an incoming edge. We hope that now an incoming edge of v' connects to another f-tree, but of course we may need to try another trade. This process terminates when we either succeed in connecting to another f-tree or decide that none of the possibilities work out.

Note that we are looking for a sequence of edge trades; we refer to this sequence as an *augmenting path*. (This augmenting path is in some sense analogous to the augmenting paths of the Ford-Fulkerson maximum flow algorithm, but they are not the same.) We now give pseudocode for the high level work of Gabow's algorithm:

```
PackTrees(G, a)
  repeat:
    initialize a new forest of f-trees, with each vertex in its own f-tree
    while there is more than one f-tree
      mark all f-trees active, except the one containing a
      while there are active f-trees
        pick an active f-tree and search for an augmenting path
        if the search fails, return the set of trees found so far
        else do the sequence of trades, and mark inactive both of the f-trees involved
    add the one f-tree to the set of trees found so far
```

The real trick of Gabow's algorithm is to search for an augmenting path in an efficient manner. Intuitively, it should be possible to consider every edge at most once, so it should take no more than $O(m)$ time, but that means it could take $O(nm)$ time per tree, and $O(\lambda nm)$ time total. With such a bound we would not gain much by considering tree packings instead of maximum flows. However, Gabow shows how to find many augmenting paths in $O(m)$ time, such that the time to add a tree is only $O(m \log n)$. The remainder of this section describes how to find augmenting paths efficiently; readers not interested in such details may wish to skip on to Karger's algorithm.

The searches are done in a breadth-first manner. We keep track of the candidate augmenting paths by labeling each edge with the previous edge in the path. So when we start considering an f-tree root, we give all of its incoming edges a null label and add each to a queue. As a general step, we take the first edge from the queue and consider adding it to another tree. Adding an edge to a tree creates a cycle, so we label each edge of the cycle with the current edge and add it to the queue (we make sure that we only label each edge once per "round", so there is no question of relabeling edges). We also label unused incoming edges of nodes on the cycle with the cycle edge incoming to that node. If we ever find an edge that connects two f-trees we halt, and trace back along the labels to figure out how to update.

The efficiency of the above depends on several things. First, when we take an edge from the front of the queue, we do not try adding it to all the other trees. Instead we cycle through the trees, moving on to tree i when we first find an edge from tree $i-1$ on the front of the queue. Second, we process the f-trees in rounds, where each edge can be labeled at most once in a round. In particular, we start a round by marking each f-tree (except the one containing a) as active. We then pick an active f-tree, search for a way to connect it to another f-tree, make the connection and mark both trees inactive. The point is to avoid looking at an edge too many times. Efficiency here also depends on the fact that the labeling is ordered so that the labeled edges in any tree form a subtree, making it easy to look at only the unlabeled edges in a cycle. (Given that the labeled edges form a subtree, we can keep track of its root. Now given the endpoints of an edge, we figure out which endpoint is labeled and jump to the root of the labeled subtree. We now either find that the unlabeled vertex has a labeled ancestor, in which case we label down to the unlabeled vertex from there, or we label from the root of the labeled subtree up to the least common ancestor of the two vertices, and down to the unlabeled vertex. In any case we preserve the property that the labeled nodes form a subtree, and it is easy to identify the new root of the labeled subtree if it changes.)

```
Search(v)
    initialize an empty queue, Q
    label all of v's incoming edges with a special symbol start, and add them to Q
    set i to 0
    while Q is not empty
        take the first edge, e = {w, x}, off of Q
        if e is in tree i, set i to i + 1 (modulo the number of trees)
        if both w and x are labeled, continue (the while loop)
        let u be whichever of w and x is unlabeled (one must be)
        let r be the root of the labeled subtree
        repeat:
            if u = r
                for each edge f on the path from the root of the labeled subtree up to u
                        and then down to whichever of w and x is unlabeled
                    Label(f, e)
                    if the call to Label() returns an edge, return it
                break (the repeat loop)
            if u and v are in different subtrees return e
            let y be the deeper of u, r
            if y's parent edge is labeled
                for each edge f on the path from u down to whichever of w and x is unlabeled
                    Label(f, e)
                    if the call to Label() returns an edge, return it
                break (the repeat loop)
            set whichever of u and r is y to y's parent
```

```
Label(e, l)
    label e with l
    if e's head has no labeled edges
        for each unused incoming edge of e's head, f
            if f connects two f-trees, return f
            label f with e
    return NIL
```

Assuming that this method is all correct, which is not obvious, but is proved by Gabow, it is not difficult to see the time bound. Each round looks at each edge at most once, and reduces the number of f-trees by at least a factor of two. Thus it takes at most $O(m \log n)$ time to find each tree, totaling $O(\lambda m \log n)$ time.

2.3.2 Karger's Algorithm

The major problem with Gabow's algorithm is that it takes time proportional to the value of the minimum cut, which could be huge for a graph with integer edge capacities, and it does not work at all for a graph with irrational edge capacities. It is possible to give a strongly polynomial tree packing algorithm [24, 5], but the time bounds are not better than $O(nm)$. Karger finesses the problem by showing that we can get by with a tree packing in a subgraph that does have a small minimum cut.

One key observation is that a maximum tree packing must have at least one tree that uses at most two tree edges. This observation holds if we obtain a packing from a directed version of the graph, as in Edmonds' theorem, or an undirected packing, as in Nash-Williams' theorem. In the first case, each of the λ cut edges gets turned into only two directed edges, and we get λ trees, so it is not possible for all trees to contain at least three cut edges. Likewise, in the second case, we get at least $\lambda/2$ trees, so it is not possible to give at least three of the λ cut edges to each tree.

If we could find a subgraph in which the cut values were small, but corresponded to the cut values of the original graph, it would still be the case that some tree in a packing in the subgraph would use only two minimum cut edges. Even better, the argument still goes through if we only have an a subgraph in which the cut values roughly correspond to the cut values of the original graph. Given such a subgraph, we can pack trees in it and find the minimum cut by checking for cuts that only use two tree edges.

We can also motivate this approach in reverse. We know that by using Gabow's algorithm we can find tree packings quickly in graphs with small minimum cuts. Thus it would be great if we could compute a tree packing in a subgraph with small minimum cut and use it to find the minimum cut of the whole graph.

So at a very high level, Karger's algorithm is as follows:

```
K(G)
    find a subgraph G′ in which cuts correspond reasonably well to the cuts in G
    pack trees in G′
    check some trees for cuts that use at most one tree edge
    check some trees for cuts that use at most two tree edges
```

To refer to the last two steps conveniently, we say a cut k-respects a tree if it uses k tree edges. Similarly, we refer to a tree that contains only k edges of a cut as k-constraining the cut. Thus we are interested in 1 and 2-respecting cuts of a tree that 1 or 2-constrains the minimum.

We now describe how to accomplish each step.

Finding a Sparse Subgraph The subgraph can be found by taking a random sample of the edges. The following theorem of Karger captures the key property of a random sample:

Theorem 2.3.4 *[36] Consider edges with capacity greater than one to be multiple edges with capacity one. If we sample each edge independently with probability* $p = \frac{12\ln n}{\lambda\epsilon^2}$*, then with probability at least* $1-O(1/n)$ *all cut values in the sampled graph are within* $1 \pm \epsilon$ *of their expected value.*

For implementation purposes, it will be necessary to get the best constants factor we can from this theorem. We defer that chore to the implementation section, and just discuss application of this theorem here. The following argument holds with probability at least $1 - O(1/n)$. The minimum cut of the sample is at least $(1 - \epsilon)p\lambda$, because all cuts have expected value at least $p\lambda$. The edges of the original minimum cut sample down to at most $(1 + \epsilon)p\lambda$ edges. If we pack directed spanning trees, then each minimum cut edge can be used only twice, and the number of trees is equal to the minimum cut of the sample. Hence, it is impossible to have all trees use at least three minimum cut edges as long as $2(1+\epsilon)p\lambda < 3(1-\epsilon)p\lambda$. So if we take $\epsilon \geq \frac{1}{5}$, a tree packing in the sample will have at least one tree that uses only two minimum cut edges. Note that the sampled graph has minimum cut $O(\log n)$, so Gabow's algorithm will run quickly on it. By computing a sparse connectivity certificate, we can ensure that the sample only has $O(n \log n)$ edges, so in fact we can guarantee that Gabow's algorithm will run in $O(n \log^2 n)$ time.

We note that even when we do our best to get small constants, they still are not very small, and this problem becomes a sticking point in the implementation. We ended up running tests on an implementation that violated this analysis and was consistently correct, but we have not yet been able to tighten the analysis to justify this action. See Section 3.6.1 for further discussion.

Finding Tree Packings There are at least two possibilities for finding tree packings in the subgraph. One is to make our undirected graph directed and use Gabow's algorithm, which is described above.

Another possibility is to approximate a packing of undirected spanning trees with the fractional packing algorithm of Plotkin, Shmoys, and Tardos [52] (PST). This approach hast the theoretical advantage that in some cases we may find more than $\lambda/2$ trees, and in this case there will be a tree that uses only one minimum cut edge. We did not implement this method, so we do not describe it further.

Finding 1-respecting Cuts We now need a way to find the minimum cuts that use only two tree edges. We start with minimum cuts that use only one tree edge.

Define $v^{\downarrow}$ to be the descendents of v in a tree. (Assume the tree is rooted. We can root our trees arbitrarily if necessary.) For arbitrary f, define $f^{\downarrow}(v) = \sum_{w \in v^{\downarrow}} f(w)$. Given values $f(v)$ it is easy to compute $f^{\downarrow}(v)$ for all v in linear time with a depth first (postorder) traversal of the tree.

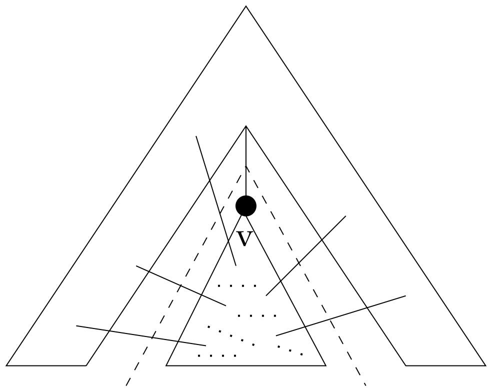

Figure 2.5: Cut defined by v. The cut edges are drawn solid. $c^{\downarrow}(v)$ counts the cut edges, as well as double counting the dotted edges (which are counted once by $\rho^{\downarrow}(v)$), so the cut value is $c^{\downarrow}(v) - 2\rho^{\downarrow}(v)$.

We notice now that $v^{\downarrow}$ defines a cut that uses only one tree edge (v's parent edge). So we might hope to compute cut values of 1-respecting cuts by computing $c^{\downarrow}(v)$. This quantity is not quite correct though. $c^{\downarrow}(v)$ counts the total capacity of edges leaving the subtree rooted at v, but double counts the capacity of edges that connect two nodes in $v^{\downarrow}$ (see Figure 2.5). The quantity we want, the cut value if we cut v's parent edge, is the total capacity of edges that leave the subtree rooted at v. To fix this problem, we let $\rho(v)$ denote the total capacity of edges whose endpoints' least common ancestor is v. All m least common ancestors can be found in $O(m)$ time, so we can compute $\rho(v)$ in $O(m)$ time. Observe that $\rho^{\downarrow}(v)$ is precisely the total capacity of edges that connect two nodes in $v^{\downarrow}$. Thus $C(v^{\downarrow}) = c^{\downarrow}(v) - 2\rho^{\downarrow}(v)$. It follows that we can find all cuts that use only one tree edge in $O(m)$ time.

Finding 2-respecting Cuts (the simple way) We can extend this approach to cuts that use two tree edges. A pair of nodes, v and w, now define a cut. We say that v and w are comparable if one is an ancestor of the other, and incomparable otherwise. If v and w are incomparable, we are interested in the quantity $C(v^{\downarrow} \cup w^{\downarrow}) = C(v^{\downarrow}) + C(w^{\downarrow}) - 2C(v^{\downarrow}, w^{\downarrow})$ (see Figure 2.6); if (without loss of generality) $v \in w^{\downarrow}$, we are interested in the quantity $C(w^{\downarrow} - v^{\downarrow}) = C(w^{\downarrow}) - C(v^{\downarrow}) + 2C(v^{\downarrow}, w^{\downarrow} - v^{\downarrow})$ (see Figure 2.7).

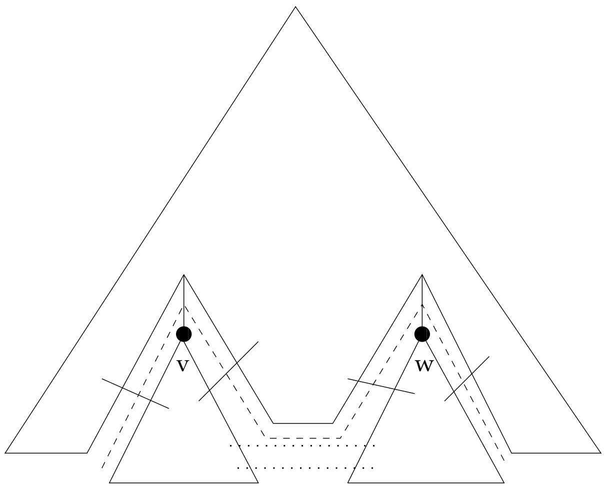

Figure 2.6: Cut defined by incomparable v and w. The cut edges are drawn solid. $C(v^{\downarrow}) + C(w^{\downarrow})$ counts the cut edges, as well as double counting the dotted edges, namely those counted by $C(v^{\downarrow}, w^{\downarrow})$. So the cut value is $C(v^{\downarrow}) + C(w^{\downarrow}) - 2C(v^{\downarrow}, w^{\downarrow})$

We already know how to compute $C(v^{\downarrow})$ and $C(w^{\downarrow})$, so we need only worry about the other terms. Define $f_v(w) = c(v, w)$, the capacity of the edge, if any, between v and w. Define $g_w(v) = f_v^{\downarrow}(w)$. As argued above, we can compute this function for any v in $O(n)$ time. So we can get all $O(n^2)$ values for pairs v, w in $O(n^2)$ time. Now $g_w^{\downarrow}(v) = C(v^{\downarrow}, w^{\downarrow})$, the desired quantity in the first case, and we can also compute it in $O(n^2)$ time. In the second case, the sum $g_w^{\downarrow}(v)$ double counts edges with both endpoints in $v^{\downarrow}$, so to get the desired quantity we just take $g_w^{\downarrow}(v) - 2\rho^{\downarrow}(v)$.

We now have an algorithm to compute all cuts that use at most two tree edges in $O(n^2)$ time.

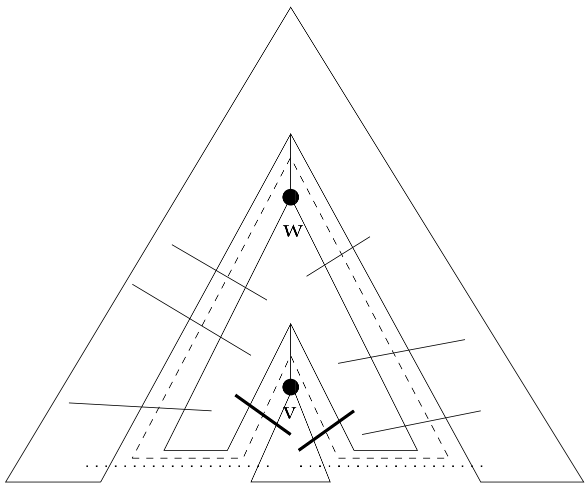

Figure 2.7: Cut defined by comparable v and w. The cut edges are drawn solid. $C(w^{\downarrow})$ counts the thin solid edges and the dotted edges. $C(v^{\downarrow})$ counts the thick solid edges and the dotted edges. $C(v^{\downarrow}, w^{\downarrow} - v^{\downarrow})$ counts the thick solid edges. Thus the cut value, the thick solid edges plus the thin solid edges, is given by $C(w^{\downarrow}) - C(v^{\downarrow}) + 2C(v^{\downarrow}, w^{\downarrow} - v^{\downarrow})$.

Assembling everything, we have a minimum cut algorithm that with probability at least $1 - 1/n$ finds all minimum cuts and runs in $O(n^2 \log n)$ time.

A Faster Way to Find 2-respecting Cuts (the fancy way) The method given above computes the cuts for all $O(n^2)$ pairs of tree edges, and therefore takes $\Omega(n^2)$ time, regardless of the input. If all we want is one minimum cut, we would hope to avoid considering some pairs of tree edges. It turns out that we can. Some readers may wish to skip these details and move on to the implementation chapter.

We start by fixing some vertex v. We now want to find the w such that the cut defined by cutting v's parent edge and w's parent edge is minimum. The key observation is that if v and this w are incomparable, then w must be an ancestor of a neighbor of a descendant of v, because we can demand that each side of the minimum cut be connected. (In order for $v^{\downarrow} \cup w^{\downarrow}$ to be connected, there must be an edge between a descendant of v and a descendant of w.) If v and w are comparable, then without loss of generality we can assume w is v's ancestor. This observation suggests a way to restrict the sums given above. We need only check vertices that fit one of the two conditions.

It is not immediately clear that we will gain anything, because v (or a neighbor of a descendant of v) may have $O(n)$ ancestors. So to be efficient, we use dynamic trees [54, 55], which among other things, support the following two operations on a tree with values at the nodes ($\mathrm{val}(w)$ at node w):

$\mathsf{AddPath}(v, x)$ add x to the value of every node on the path from v to the root.

$\mathsf{MinPath}(v)$ find the node on the path from v to the root with the minimum value.

Both operations can be done in $O(\log n)$ time.

Thus for a leaf v, we can find its incomparable "partner" w by initializing all nodes u to have value $C(u^{\downarrow})$, calling $\mathsf{AddPath}(u, -2c(v,u))$ for each neighbor u of v, and then calling $\mathsf{MinPath}(u)$ for each neighbor u of v and taking the minimum w returned. This method works because the AddPath calls result in $val(w)$ being $C(w^{\downarrow}) - 2C(v, w^{\downarrow})$. So $val(w) + C(v^{\downarrow}) = C(v^{\downarrow} \cup w^{\downarrow})$ (recall that v is a leaf, so $v^{\downarrow} = v$). We want to minimize this quantity over w, for which we need only look at $val(w)$, and this is precisely what the calls to MinPath do.

```
LeafIncomparablePartner(v)
    for all u, val(u) ← C(u↓)
    for all edges {v, u}, AddPath(u, −2c(v, u))
    return the w corresponding to the minimum value over edges {v, u} of MinPath(u)
```

We can find v's comparable partner by using the same initialization, calling $\mathsf{AddPath}(u, 2c(v,u))$ for each neighbor u of v, and just checking $\mathsf{MinPath}(v)$. This time the calls to AddPath result in $val(w) = C(w^{\downarrow}) + 2C(w^{\downarrow} - v, v)$. So we get the desired quantity by subtracting $C(v^{\downarrow})$, and we can find the minimum over comparable w by the call to MinPath. Note that there is only one call to MinPath, because in this case we are only interested in ancestors of v.

```
LeafComparablePartner(v)
    for all u, val(u) ← C(u↓)
    for all edges {v, u}, AddPath(u, 2c(v, u))
    return the w corresponding to MinPath(v)
```

Note that we cannot afford to actually initialize the dynamic trees for each v we process. We have to initialize them once and then undo dynamic tree operations when we are done processing a given leaf. At least in theory, this is not a problem.

To process an internal node, we need to look at all the neighbors of its descendants. The basic idea is to process all the leaves of the tree and contract them into their parents. We can then process the leaves of the new tree. If the tree is balanced, $O(\log n)$ such phases suffice to process the whole tree. Unfortunately, it is possible to have n nodes with downward degree one, in which case we do $O(n)$ phases.

However, after we process the only child of a node v, we can immediately process the node by doing AddPath operations for each of its neighbors and then MinPath operation(s). We already have the values set for the neighbors of the descendants, so the AddPath operations will update the values appropriately. The partner of v is then either the best partner found for the descendant, or an ancestor of a neighbor of v. Note that we cannot immediately process a node with downward degree two, because after processing one child we must undo those AddPath operations before we can process the other child. We then have to redo them when we process the node itself. It is because no undoing is necessary for a node with downward degree one that we can go right ahead and process it.

It follows that we can process in one phase all nodes that have downward degree less than two and do not have as descendants any nodes with downward degree more than one. After a phase, each new leaf node is a node that had at least two children previously (or else it would have been processed), so the number of leaves decreases by a least a factor of two in each phase. It follows that we need only $O(\log n)$ phases. Each phase does at most $O(1)$ dynamic tree operations per edge, and each dynamic tree operation takes $O(\log n)$ time, so the time to process the whole tree is $O(m \log^2 n)$.

Chapter 3

Implementation

Unfortunately, knowing how an algorithm works in theory is not the same as knowing how to implement it well. In this chapter we discuss the important implementation details of the four algorithms, including choice of data structures, implementation of primitives, choice of subroutines, settings of parameters, and heuristic improvements.

In general, we only concern ourselves with implementation details that have major impact. It is almost always possible to speed up a program a bit by cleverly twiddling the code, but those are not the sort of changes that concern us. The rule of thumb we adopted was not to worry much about any details that changed the runtime by less than a factor of two. Some heuristic improvements affected the runtime by factors of a thousand; these are clearly of much greater interest.

Another rule we use in the implementations is that a heuristic be amortized against the running of the underlying algorithm. In other words, a heuristic run periodically should not cost more than the work done by the algorithm in the interim. Further, any heuristic preprocessing should not take more than near linear time, as linear work is all that we can be assured the algorithm will need to do. The point is that while we wish to get as much benefit as we can from heuristics, we do not want it to ever be the case that an implementation suffers terribly on some problems because of its use of heuristics. For example, we would not allow a preprocessing heuristic that ran in quadratic time, because even if the worst case running time of the underlying algorithm is cubic, it is always possible that it will run in linear time. If this happens, then we would lose a great deal by running the heuristic. On the other hand, after the algorithm has done quadratic work, there is no harm in running the heuristic, because even if it fails we lose no more than a factor of two in total running time. Our strategy guarantees that failed heuristics never dominate the runtime, and that successful heuristics are not much more expensive than the underlying algorithm.

Throughout this chapter we mix discussion of the abstract algorithms and our implementations of them. To avoid confusion we distinguish them by typeface. Sans serif font refers to an algorithm: HO, NI, KS, K. Typewriter font refers to an implementation: `ho`, `ni`, `ks`, `k`. Since we have many variants of each implementation, we will distinguish them with suffixes. For example, `ho_nopr` refers to an implementation of HO that does not include PR heuristics.

We begin by discussing graph data structures and implementation of the contraction operation. Next we discuss general issues in incorporation of the Padberg-Rinaldi heuristics. Then we discuss each of the algorithms in turn.

3.1 Graph Data Structures

The internal representation of the graph is very important. There are many possible choices, and designation of the "best" one is only possible with respect to the operations that must be supported.

HO and NI clearly need a representation that makes it easy to find the neighbors of a given vertex, in order to support the push/relabel operations and the graph search, respectively. Gabow's algorithm also clearly needs an adjacency representation, so K needs one if it uses Gabow's algorithm to do the tree packing. K will also need one if it uses the fancy (dynamic trees) approach to finding 2-respecting cuts. If K is implemented without either of these subroutines, as it can be, then it is less clear what data structure is right. For KS, especially in the variant we give, there is no apparent reason why it would need an adjacency structure. In fact, what we really want for KS is to be able to pass over all of the edges quickly so we can decide whether to contract each of them.

So `ho`, `k`, and `ni` all represent an undirected graph as a symmetric directed graph using the adjacency list representation. Each vertex has a doubly linked list of edges adjacent to it and pointers to the beginning and the end of the list. An edge $\{u, v\}$ is represented by two arcs, (u, v) and (v, u). These arcs have pointers to each other. An arc (u, v) appears on the adjacency list of u and has a pointer to v.

We note that another possibility would be an adjacency matrix; however, this representation would be very space inefficient for a sparse graph. The space could be reduced by hashing, but hashing removes the simplicity of an adjacency matrix that makes it attractive. We did not explore this possibility.

`ks` represents an undirected graph as an array of edges. This representation has less flexibility than the adjacency list, but KS really does not need it, so a smaller and simpler representation makes sense. It is possible that it makes little difference.

3.1.1 Implementing Contraction

As mentioned in Section 2.2, contracting nodes v and w consists of merging v and w. This action will create parallel edges if v and w have any neighbors in common, and will create a self-loop if v and w are neighbors. Parallel edges can be merged into one edge with capacity equal to the sum of the original two capacities; self-loops can be deleted. So we have two important implementation issues: how to represent merged vertices, and what to do about unnecessary edges.

One possibility is to do the contractions explicitly, so that a node is always represented the same way as a vertex, without self-loops or parallel edges. We refer to this strategy as *compact contraction*, and implement it as follows:

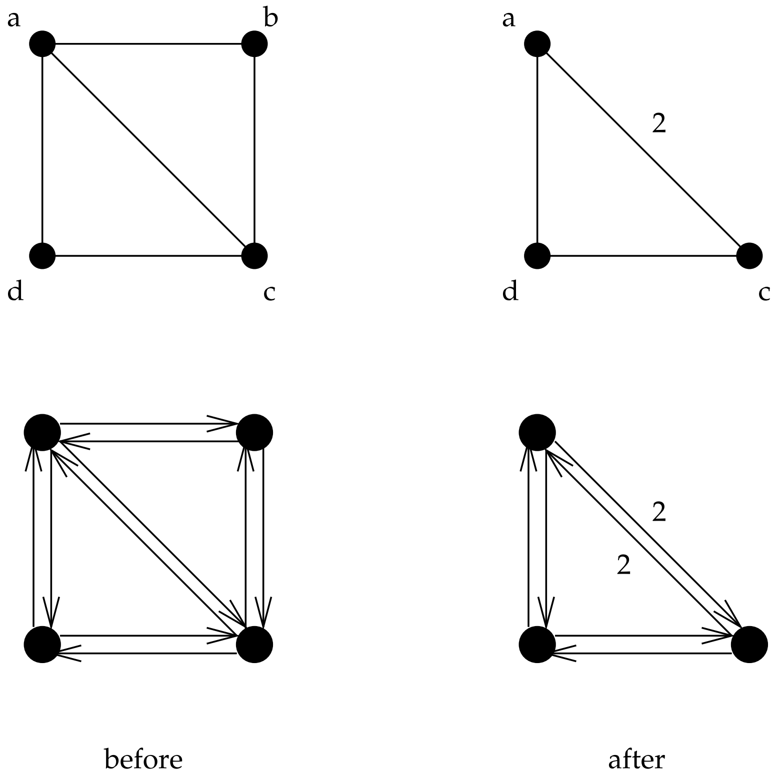

Figure 3.1: Compact contraction example: Contracting edge $\{a, b\}$ in a graph represented by adjacency lists. Graphs in the top row are represented as shown in the bottom row. If an arc capacity is equal to one, the capacity is not shown.

```
CompactContract(v,w)
    Replace each edge {w, x} with an edge {v, x}.
    Delete w from V(G).
    Delete self-loops and merge parallel edges adjacent to v.
```

Figure 3.1 gives an example of this implementation of edge contraction for the adjacency list representation of a graph.

With the graph represented by an adjacency list, a careful implementation of compact contraction of v and w takes time proportional to the sum of degrees of v and w before the contraction. With the graph represented by an array of edges, compact contraction can take $O(m)$ time, and therefore is probably not practical.

Another possibility is to implicitly represent nodes by the sets of vertices they contain. We do not need to actually merge v and w, we just need to be able to tell what node a vertex belongs to. This observation suggests using a disjoint-set union data structure to keep track of the nodes of the contracted graph. We refer to this strategy as *set-union contraction* and implement it as follows:

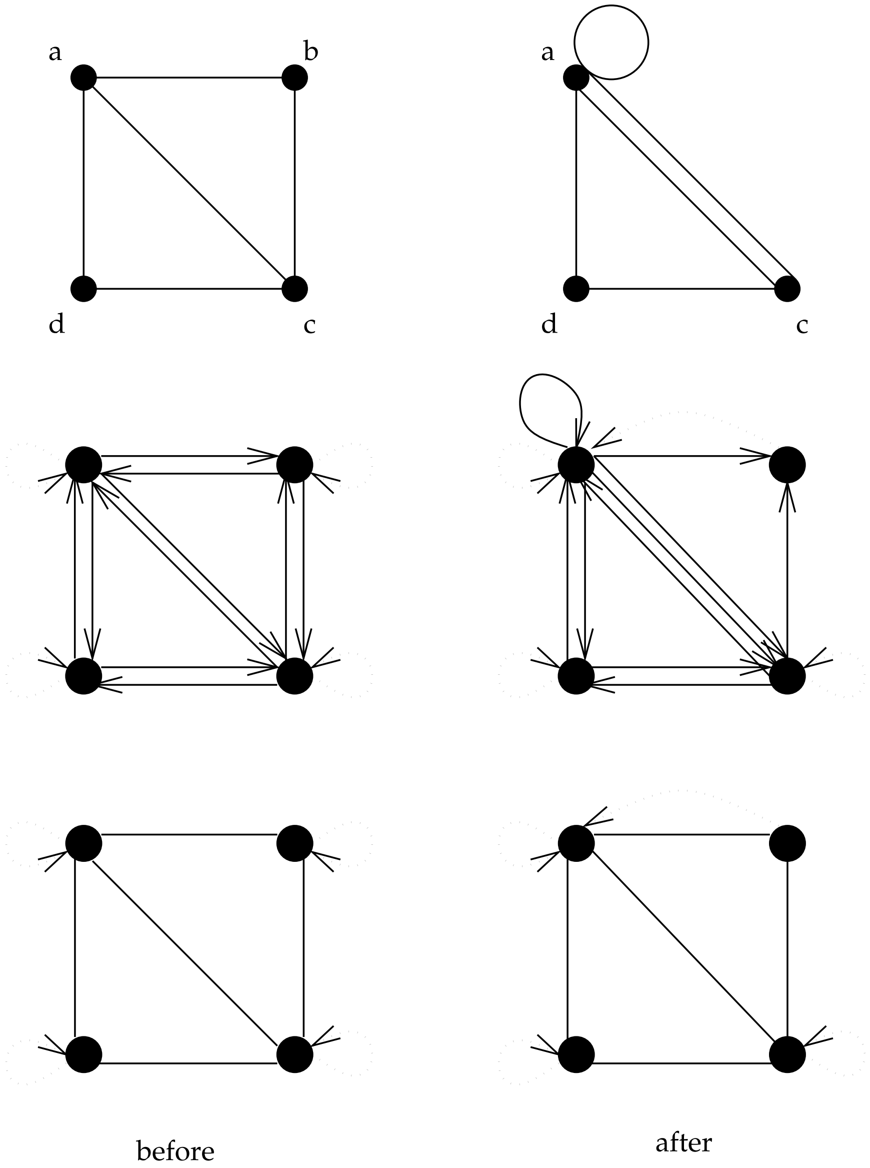

Figure 3.2: Set-union contraction example: Contracting edge $\{a, b\}$. Graphs are in the top row. An adjacency list representation is in the middle row, and an array of edges representation is in the bottom row. Dotted lines are pointers for the set-union data structure.

```
SetUnionContract(v,w)
    union(v,w)
```

We implement the disjoint-set union data structure by disjoint-set forests with path compression; see *e.g.* [14]. In this representation each set has a distinct representative, each vertex has a pointer towards (but not necessarily directly to) the representative of its set, and representatives point to themselves. We assume that the reader is familiar with the set-union data structure.

The advantage of set-union contraction is that a contraction takes only $O(\alpha(n))$ time, where α is a functional inverse of Ackermann's function (bounded by 4 for n less than the number of particles in the universe). Disadvantages come from the parallel arcs and self-loops which remain in the graph. With parallel arcs, operations that we would have done in one step could take many steps. Worse, we are not guaranteed that the number of arcs in a contracted graph is $O(n^2)$. The other disadvantage is that it now also takes $O(\alpha(n))$ time to find the head node of an arc. Note however, it is possible to use set-union contraction to do many individual contractions and then compact the graph in one pass. We call this operation *compaction*. This approach keeps the benefit of fast contractions and hopefully cleans up before the parallel arcs get out of hand. We implement compaction as follows:

```
CompactGraph(G)
    compute the set representative for every edge endpoint
    if the graph is represented as an edge array
        sort the edges by endpoints
        pass over the sorted list of edges, combining parallel edges and removing self-loops
    else (graph is represented by edge lists)
        append the edge lists of each vertex to the list of its set representative
        for each set representative
            unmark any marked neighbors
            for each edge e
                if e is a self-loop, delete it
                else if e reaches an unmarked neighbor, mark the neighbor with e
                else merge e with the neighbor's mark
    remove all vertices that are not set representatives
```

Using two calls to a counting sort algorithm to implement the sort step, compaction can easily be done in $O(m)$ time with either graph representation.

We make use of all of these strategies at different times. In general we use set-union contraction followed by a compaction when we have many contractions to do, and we use compact contraction when we have few contractions to do. We will discuss this issue further in subsequent sections.

3.2 The Padberg-Rinaldi Heuristics

It is not clear how to get the most benefit from the PR tests. One natural strategy, which is used by Padberg and Rinaldi [51], is to keep applying them until no edges pass any of the tests. We refer to this approach as the *exhaustive* strategy. The problem is that this strategy takes too long. It could take $\Omega(mn)$ time just to apply tests PR3 and PR4 to every edge. Since edges change during contractions, and therefore must be tested again, if one PR test passes at a time, applying these two tests exhaustively could take $\Omega(n^2m)$ time, dominating the running time of all the algorithms we are implementing. Even PR1 and PR2 could take $\Omega(mn)$ time to apply exhaustively. So our rule on the cost of heuristics removes the exhaustive strategy from consideration.

An alternative strategy is to apply the tests only to edges that change. Recall that a contraction can create parallel edges, which may be merged. The resulting edge has larger capacity, so a PR test may now apply, even though it did not apply to either edge individually. We refer to this approach as the *source* strategy, because in a contraction based implementation of GH (or HO), which contracts the source and sink at the end of a max-flow computation, edges incident to the source are the ones that change. This strategy also extends to NI: apply the PR tests near the edge contracted after the last search. Both Padberg and Rinaldi [51] and Nagamochi et al [48] make use of this approach. Note that it could be similarly extended to KS as it was originally described, where we contract one edge at a time, but it does not make sense for our variant, where we contract many edges at once. It also makes no sense for K, which does not fit into the framework of GenericContractCut. Note that this strategy does not replace the exhaustive strategy, because we apply the tests where the underlying algorithm has changed the graph, but then do not reapply according to where the tests change the graph. (If we did, then it would be the exhaustive strategy, and we would not have gained anything.)

As an alternative to the above methods, we introduce a new approach, which we call the *pass* strategy. The basic idea is to apply the tests as much as possible in linear time. For PR1 and PR2, there is a natural way to implement this idea: apply each test once to each edge. For PR3 and PR4 we need to skip many edges. We approach this problem by making a new low-level test that applies PR3 and PR4 to all edges incident to a node. Recall that PR3 looks at an edge and the other two sides of a triangle, and PR4 looks at all triangles an edge is in. We implement our test as follows:

```
PRTest34(v)
    label all neighbors of v
    for each unscanned neighbor w of v
        sum = 0
        for each neighbor x of w that is labeled (i.e. is a neighbor of v)
            add min(c(v, x), c(w, x)) to sum
            apply test PR3 to the triangle {v, w}, x
        apply PR4 to {v, w} using the info in sum
        mark w scanned
    mark v scanned
```

The main point is that a neighbor's incident edges are only looked at if it is unscanned; it is therefore immediate that the following implementation of a pass only takes linear time:

```
PRPass(G)
    apply PR1 and PR2 once to each edge
    mark all nodes unscanned
    while there is an unscanned node v, PRTest34(v)
```

A linear time pass turned out to be very useful. For one thing, it makes sense to preprocess the input with it, because it only takes linear time. Then, if it contracted away a constant fraction of the nodes, we apply it again. We used this preprocessing strategy for all the algorithms, although we used a different constant for different algorithms. It turned out that preprocessing alone killed several otherwise interesting problem families—only two nodes would be left by the time we were done. A linear time pass also integrates nicely into NI and KS. We will discuss this issue further in subsequent sections.

Observe that this strategy needs an adjacency structure to support PR3 and PR4, but it can be implemented with either contraction method. We can either do compact contractions or do set-union contractions and compact when we are done. The former has the advantage that parallel arcs are merged right away, which may cause a later test to pass when it would not have otherwise. The latter has the advantage that one compaction at the end takes only $O(m)$ time, whereas many compact contractions could take $O(n^2)$ time. Our experience was that compact contraction was better for tests PR3 and PR4, because it caused more tests to pass. Set-union contraction was typically a bit faster than compact contraction for PR1 and PR2, but there was less difference in the number of passed tests.

We note that a subtlety of the tests says that we must not involve any v in more than one PR2 or PR3 contraction met with equality in one pass if we use set-union contraction. Recall that in this case $\{v\}$ may be the minimum cut, and once we have done one set-union contraction involving v, the old $c(v)$ will not be correct. An easy example where we would get into trouble is a line of five nodes, with edge capacities two, one, one, and two. PR2 applies to both of the edges with capacity one, but after contracting one, it does not apply to the other. If we do both contractions blindly, we will miss the minimum cut.

Note that for the purposes of explanation we have left out some details from the above pseudocode. In particular, one must be careful about how nodes are labeled in PRTest34 so there is no confusion between different calls, and obviously one must not apply a test to an edge that has already been contracted.

We have also failed to specify in what order to apply PRTest34. Our experience was that over several passes it was good to try to give every node a chance to be scanned, and it was good to apply tests again where they had succeeded before. To achieve this, we assigned every node a score, initially zero. Each time it was involved in a contraction its score was incremented, and each time it was skipped for a PRTest34 its score was incremented by two. We then picked nodes in decreasing order by score. Using a random order worked reasonably well too.

We warn anyone who implements these tests that they are very subtle. Seemingly little changes do often affect performance. For example, the seemingly innocent change of applying PRTest34 in an arbitrary fixed order causes us to lose many contractions. It took us a long time to arrive at the strategies described here. We recommend comparing any new strategies against the ones we use.

3.3 Hao-Orlin Algorithm

We based our implementation `ho` on the push-relabel max-flow code of Cherkassky and Goldberg [13]. Thus several implementation decisions were made based experience from the max-flow context. Such decisions are likely appropriate, but not above question.

We begin by discussing choices we made in the implementation that are not addressed by the algorithm. We then discuss the heuristics we added.

3.3.1 Basics

Graph Representation As mentioned previously, we need an adjacency data structure for HO, so we use adjacency lists. Contractions tend not to happen in groups, so we use compact contraction everywhere in `ho`.

Push-Relabel Strategy We chose to use the highest-label strategy to pick which nodes to discharge first. This strategy seems to give the best results in practice in the max-flow context [13]. For this reason we did not consider the fancy approach that uses dynamic trees.

We use an array of buckets $B[0 \ldots 2n-1]$ to implement this efficiently. Bucket $B[i]$ holds both a list of all awake vertices with distance label i and a list of those that are also active. This makes it easy to keep track of the highest label active node, as well as making it easy to detect nodes that have become disconnected from the sink and must be put to sleep.

Source and Sink Selection In some cases the algorithm is sensitive to the way the first source-sink pair is chosen. We chose a vertex with the largest capacity as the first sink and a neighbor of this vertex as the first source. Brief testing suggested that this choice works well in general.

3.3.2 Heuristics

Global Updates In the maximum flow context, it is useful for many problem classes to periodically compute exact distances to the sink. This operation is known as a *global update*, and it can easily be accomplished by a backwards breadth-first search from the sink. (Backwards means that we traverse the directed edges in the wrong direction, which is easy, because we also have arcs in both directions.) We make several natural modifications in order to use this idea in HO.

- The sink's distance label is nonzero, so the backwards breadth-first search starts with the sink's distance label instead of zero.
- The computation is done only on the awake graph. (We do not want to disturb the sleeping nodes.)
- Vertices not reachable by the breadth-first search computation are put to sleep without changing their distance labels. (The sink is not reachable from these vertices.)

In order to ensure that we do not spend too much time on global updates, we explicitly amortize against relabels: a global update is performed as soon as the number of relabels since the last global update exceeds β times the number of awake vertices. Thus relabeling time always dominates time spent doing global updates. In our implementation, $\beta = 2$.

Global updates do not always improve the running time; on some problem families, such as graphs with heavy components¡, the running times become worse. However, the running times never become much worse in our tests, and sometimes are much better than without global updates.

PR Heuristics As in all our algorithms, we preprocess with PR tests before doing anything else. We also make use of the PR tests during the execution of HO, which is very tricky, because HO reuses flow information for successive flow computations. We have to be careful not to disturb the flow information when we do contractions as a result of the PR tests.

Fortunately, it is easy to show (using [31]) that it is safe to contract an edge incident to the source, as long as we saturate any new outgoing capacity from source that this operation creates. So we can safely use the source PR strategy. Using the source strategy is also appropriate, as it is the edges near the source that typically change. Unfortunately, the algorithm frequently does little work in a flow computation, and the source quickly becomes high degree, making even a source test expensive. Thus we explicitly amortize against the work of the algorithm: we apply a source test when the algorithm has done enough work in pushes and relabels to dominate the cost of the last test.

Note that it is possible to do a PR pass, which might contract an edge not adjacent to the source, but we must do a global update immediately afterwards to restore the validity of the distance labeling. Our experience with this idea was that it usually just slowed down the code, so we stopped using it.

Excess Detection We introduce a simple heuristic that often allows us to contract a vertex in the middle of a flow computation. The general results on the push-relabel method [26] imply that the excess at a vertex v is a lower bound on the capacity of the minimum s-v cut. Thus, if at some point during an s-t cut computation the excess at v becomes greater than or equal to the capacity of the minimum cut we have seen so far, we contract v into the source and saturate all the arcs going out of v. Note that v can be either awake or asleep. The correctness proof for this heuristic is straight-forward. We call this heuristic *excess detection*.

A special case of the excess detection heuristic occurs when v is the sink. In this case we can stop the current s-t cut computation and go on to the next one. (Remember that we do not actually care about the s-t cut unless it is a smaller cut than we have seen before, and the fact that excess detection applies means that it is not, so we can contract s and t and move on.)

Excess detection is inexpensive and on some problems, it reduces the number of s-t cut computations significantly. We note that one needs to be careful when implementing excess detection, because one contraction can cause excess detection to pass at other nodes. (Suppose the excess at v becomes large and v is contracted into the source. When v's outgoing arcs are saturated, excess at some of v's neighbors may become large, and these neighbors should be contracted as well.)

This problem can be handled by keeping track of the nodes waiting to be contracted in either a stack or a queue, as long as we make sure not to put a node on the stack/queue more than once. In general, excess detection requires care, because we are changing the graph during a maximum flow computation.

Note that in fact the total flow entering v is a lower bound on the minimum s-v cut, and the total flow is always at least as large as the excess, so we could get a stronger test by using this quantity. However, we already need to maintain excesses, whereas we do not need to maintain total incoming flow, and brief testing suggested that the extra cost of maintaining this information negated the benefit from the contractions gained.

Single-Node Layers Suppose a sleeping layer consists of a single node v. When this node is awakened it will be necessarily be the only awake node other than the source, so it will become the sink, and the flow value will be its excess plus any further flow it can get directly from the source. Further, we know that all nodes that are awake at the time v is put to sleep will be contracted into the source by the time it is awakened. Thus the "further flow it can get directly from the source", is precisely the total capacity from the currently awake nodes. We therefore have all the information to compute the s-v flow at the time v is put to sleep, so instead of bothering to put it to sleep we compute the flow value and contract it into s immediately. This reordering can be helpful because it may cause PR tests and/or excess detection tests to pass earlier than they would have.

3.4 Nagamochi-Ibaraki Algorithm

NI required relatively little modification from its theory description. We just incorporated two heuristics given by Nagamochi et al. [48], made some careful data structure choices, and incorporated PR tests. We differ from [48] on the latter two points.

Nagamochi-Ono-Ibaraki Heuristics Nagamochi et al. [48] give a heuristic modification to NI that often helps to update the upper bound on the minimum cut. The heuristic takes advantage of the fact that the set of visited nodes is connected, and therefore defines a cut. It may seem that this is just an arbitrary cut, but recall that we always pick the most connected node to visit next. So for example, if we have two cliques connected by a single edge, we will likely visit all of the nodes of one clique before visiting any nodes of the other. Thus if we always check the cut defined by the set of visited nodes, we will find the minimum cut. Furthermore, since most of the edges in this graph are unnecessary, as soon as we find the minimum cut the sparse certificate will allow us to contract most of the edges. In general, this heuristic is very helpful at allowing us to get the most out of our sparse certificates.

We can easily keep track of this cut value by adding $c(v)$ and subtracting $2r(v)$ each time we visit a node v. (Recall that $r(v)$ is the capacity of edges between v and the visited nodes, so the previous cut value contains $r(v)$ once. Adding $c(v)$, we count $r(v)$ again and add in the capacity from v to unvisited nodes. So subtracting $2r(v)$ we get the desired quantity.) We use the value to update our cut upper bound. The code for this heuristic, called the α heuristic, appears at line (*) below.

We now give the pseudocode for a scan-first search that contracts the contractible edges it finds. Note that this is different from the description in the theory chapter in that we do not explicitly build the spanning forests, and we do not assume that edges are uncapacitated.

```
ScanFirstSearchContract(G, λ̂)
    for each v ∈ V
        r(v) ← 0
        mark v unvisited
    for each e ∈ E
        mark w unscanned
    while there is an unscanned node
        v is the unscanned node with largest r(v)
(*)     α ← α + c(v) − 2r(v)
        λ̂ ← min{α, λ̂}
        for each unscanned e = {v, w}
            r(w) ← r(w) + c(v, w)
            if (r(w) ≥ λ̂)
                G ← G/(v, w) with new node v′
                λ̂ = min{c(v′), λ̂}
            Mark e scanned
        Mark v visited
    return λ̂
```

Priority Queue The theory bound given by Nagamochi and Ibaraki depends on use of a priority queue with a constant time increase-key operation, *e.g.* a Fibonacci heap. Preliminary experiments suggested that Fibonacci heaps do help a little bit on very dense graphs, but otherwise it is better to use a (simpler) k-ary heap. In the end we chose to use a 4-ary heap. Note that this makes the theoretical worst case time bound on our implementation $O(mn \log n)$.

PR Heuristics Nagamochi *et al* [48] incorporate the PR tests by applying a source test at the node created by the last contraction of a search. This approach turns out to be very helpful, but their implementation has disadvantages: there is no preprocessing stage, the tests are only applied to one node after each search, and they are not careful to make sure the tests only take linear time each time.

So we add preprocessing, and we do a PRPass (Section 2.2.1) at the end of every kth search. (We used $k = 2$.) Since a pass takes only linear time, and a search takes slightly more than linear time, we respect our rule on the time consumed by heuristics. It also turns out that this strategy is very effective. We use the score method described in Section 3.2 to decide which nodes to test, but we clear the scores after each search, so that we always test first the nodes involved in the most contractions during the search. Note that we also perform the source test, as we do not necessarily test the result of the last contraction in a pass, and sometimes the source test is very helpful.

Graph Representation As mentioned previously, we need an adjacency structure, so that is what we use. Implementation of contraction is trickier. We implemented NI using both compact contraction and set-union contraction. Preliminary experiments showed that the two data structures are incomparable in practice: each was significantly faster on some problems and significantly slower on others.

The problem is that we are only guaranteed one contraction per search, in which case we would prefer to use compact contraction. It is possible however, that we do many contractions, in which case we would prefer to use set-union contraction. So we can use set-union contraction and compact periodically, but we also have the problem that the PR heuristics prefer compact contraction. So we adopt the strategy of using set-union contraction during the scan-first search, compacting at the end, and using compact contraction during PRPass. This approach never allows the parallel edges to get out of hand, allows the PR tests to use their preferred method, and is never too expensive—even if we do few contractions, the cost is usually less than that of the preceding search.

Our final high level implementation is as follows:

```
NI(G)
    λ̂ ← min_v c(v)
    while G has more than two nodes
        λ̂ ← ScanFirstSearch(G, λ̂)
        CompactGraph(G)
        if this is a kth (second) iteration
            do source PR tests at last node involved in a contraction
            PRPass(G)
    return λ̂
```

3.5 Karger-Stein Algorithm

For KS, the main implementation decision we made was to implement our variant (Section 2.2.3) instead of the original version. Early testing suggested that our variant was better, but now that we understand the algorithm better it would be interesting to implement the original, to make sure that our variant was a good idea. Note that we violate our rule about heuristics here, because we have not been able to prove that the variant is as fast as the original in the worst case, so more extensive comparisons to the original should be done.

We needed to use an exponential distribution to sample each edge with probability exponential in its capacity. Though we were initially concerned by the resulting large number of calls to exponential/logarithm functions, we found that in practice the generation of these random numbers was not a significant part of the running time. Beyond that we just needed to deal with data structures and the PR heuristics.

Graph Representation As mentioned previously, KS does not really need adjacency lists, so we did not use them. We use set-union contraction to do all the contractions of a phase and then compact the structure. With a careful implementation of this approach, it is relatively easy to undo the contractions when we back out of the recursion. (It would have been more difficult with adjacency lists; the best thing would probably be to just copy the graph.)

PR Heuristics The decision not to use adjacency lists basically rules out easy use of PR3 and PR4. Since it is clearly desirable to have the same preprocessing as the other codes, we actually input the graph in `ni`'s data structures, ran the preprocessing, and then switched to the array of edges.

We experimented with internal PR3 and PR4 tests, because we were concerned about not having them, but we found that they were of little help. A possible explanation for this effect is the following (we focus on the PR4 case; a similar arguments applies for the PR3 test). The PR4 test applies when the sum of capacities on length-two paths exceeds λ, where the capacity of a length two path is the smaller of its edge's capacities. Consider a randomized contraction phase and its impact on such a "PR4 structure". The total capacity of edges in the PR4 structure is 2λ, implying that the probability some edge in the structure is contracted exceeds $3/4$. Especially over multiple levels of recursion, this accumulates much faster than the $1/2$ chance that a minimum cut edge will be contracted. Once we contract an edge in the PR4 structure, the PR4 test will no longer apply. In other words, in an intuitive sense, randomized contraction is taking care of the PR4 and PR3 tests before we have time to apply them explicitly.

It remains to describe our internal use of PR1 and PR2. Since we already pass over all the edges of the graph, contracting them with an appropriate probability, it is easy to incorporate PR1 and PR2. We apply the tests when we make the random choice of whether to contract an edge; we then contract it if either a test says to or the random choice says to. This implementation clearly adds only a small overhead.

As mentioned in the discussion of the tests, since we are using set-union contraction we also need to be careful not to let a node be involved in more that one PR2 test met with equality in one pass.

3.6 Karger's Algorithm

Karger's algorithm leaves open a large number of implementation options. We begin with the familiar topics of graph representation and the PR tests, and then consider each of the three parts of the algorithm in turn. One of our implementation decisions invalidates the algorithm's proof of correctness; see the section on picking ϵ for more details. Thus this implementation is actually one great big heuristic.

Graph Representation As mentioned before, it would be conceivable to implement K without an adjacency structure, but we did not attempt it. The basic graph representation is as in `ho` and `ni`. There is, however, another issue: we must represent the trees of the tree packing. Since one (capacitated) edge can occur as many tree edges, we used more adjacency list structures to represent the trees. Each tree edge maintains a pointer to the graph edge it derived from. Note that we need this adjacency structure for Gabow's algorithm; with the Plotkin, Shmoys, Tardos packing algorithm it is possible to represent the trees with a only a parent pointer for each vertex.

PR Heuristics Naturally, we also used the PR preprocessing for `k`. Since the algorithm does not do any contractions, the only way the PR tests might do further good is if we get a better upper bound on the minimum cut. If we get a new upper bound early enough in the execution so that the further contractions might help, we run the preprocessing again. We discuss how we get new upper bounds in subsequent sections. Our general finding was that our initial upper bound estimates were good enough that there was not much to be gained by doing this.

3.6.1 Sampling

There are several problems with the simple theoretical description of the sampling step that need to be finessed in an implementation.

Estimating the Minimum Cut In order to perform the sampling step correctly, the algorithm needs to estimate the value of the minimum cut. In [34], Karger gives two ways to resolve this problem. The first is to run Matula's linear time $(2 + \epsilon)$-approximation algorithm to get a good estimate (see Section 2.2.2). The other option starts by getting a crude approximation and then samples the edges, finds a tree packing, and doubles the sampling probability, repeating if the number of trees in the packing is smaller than expected. Since doubling the sampling probability doubles the number of trees, finding the final tree packing dominates the time of finding all the others. If the crude approximation was within a factor of n, then the time spent sampling is at most $O(m \log n)$.

Our experience is that it is better to run Matula's approximation algorithm. The main reason is that running Matula's algorithm allows us to compute a sparse $((2 + \epsilon)\lambda)$-connectivity certificate on the input. We can then contract all remaining edges, which can greatly reduce or even solve some problems. Furthermore, once our input graph is sparse, our sampled graph will be sparse; in particular, it will have only $O(n \log^2 n)$ edges. The tree packing step turns out to be expensive, so it is helpful to have as few edges as possible in the sampled graph.

Sampling from Capacitated Graphs Another concern is sampling a capacitated edge. In theory, we treat a graph with integer capacities as an uncapacitated graph with multiple edges, but we do not want to actually flip a coin $c(v, w)$ times for edge $\{v, w\}$, and we still do not know what to do with irrational capacities. For integer capacity edges, what we want is to pick a number from 0 to $c(v, w)$ according to the binomial distribution. Notice that the sampling probability is inversely proportional to the cut value. So if we were to multiply all the edge capacities by some large factor, causing the minimum cut to go up by that factor, the probability would go down such that the mean of the distribution would stay the same. Therefore we can approximate the binomial distribution with the Poisson distribution, which is very close to the binomial for large numbers and small mean. Picking a number according to the Poisson distribution can be done such that the number of random numbers we need is the same as the value we output (see [42]);

since the expected value of a sampled edge is always at most $O(\log n)$, this method allows us to sample using $O(m \log n)$ random numbers, regardless of the magnitude of the capacities. Since an irrational number can be approximated arbitrarily well by a rational, and we can multiply up a rational to get an integer, in the limit this method properly samples irrational capacity edges. Note that we do not need to actually carry out this process of multiplying up edges, because all we need to know to sample from the Poisson distribution is the mean, which is unaffected.

Picking ϵ Another problem is picking the ϵ used to compute the sampling probability. Unfortunately, even after reworking the analysis, the constants are quite large. Even in the limit as n goes to infinity, we end up needing to pack and check $36 \ln n$ trees (see below). We can get several trees that 2-respect (so that we do not have to check all the trees) by packing more trees, but our experience was that the time spent finding the trees was enough to make the running time several orders of magnitude worse than the other algorithms. We discovered, however, that on our test examples, finding only $6 \ln n$ trees and checking only 10 of them for 2-respecting cuts gave the right answer all the time. It is plausible that the analysis is not tight, but since we have not been able to tighten it, this implementation must be considered heuristic. There is no proof that it will be correct with the desired probability in all cases. This modification is in contrast to the other algorithms, where our heuristic changes did not affect correctness.

For reference, we now give a reworking of Karger's analysis [34] that gives the best constants we know how to get. Some readers will want to skip this section.

Recall that we sample each edge independently with probability p, and we need to bound the probability that any non-minimum cut samples to less than $(1-\epsilon)p\lambda$ edges. For any given cut, we can easily argue such a result holds with polynomially small probability by application of Chernoff bounds, but there are exponentially many cuts, so a simple union bound will not work. Fortunately, there can only be a few small cuts, and the probability of a large cut deviating is smaller than that of a small cut. Balancing the size of the cuts against the number of them, we can manage to get a result.

So this analysis depends on the number of small cuts in a graph. We refer to a cut as α-*minimum* if is has value at most $\alpha\lambda$. Unfortunately, while it is conjectured that there are only $O(n^{\lfloor 2\alpha \rfloor})$ α-minimum cuts, only special cases have been proved. We will use two of these pieces:

Lemma 3.6.1 *[34] There are at most $n^{2\alpha}$ α-minimum cuts.*

Lemma 3.6.2 *[32] For $\alpha < 3/2$, there are at most $9n^2$ α-minimum cuts.*

We assume that the program will be given a parameter f, where we are supposed to succeed with probability at least $1 - 1/f$, so we will use that parameter here.

It turns out that the first $9n^2$ cuts are the only ones of any concern. We will proceed by assuming this fact, doing the analysis, and then justifying the assumption by showing that with the constants we computed the larger cuts contribute almost nothing.

The small cuts are easy to analyze. Using a Chernoff bound on each of them and a union bound on the result, we get that

$$\Pr[\text{one of smallest } 9n^2 \text{ cuts samples to } < (1-\epsilon)p\lambda \text{ edges}] < 9n^2 e^{-\epsilon^2 p\lambda/2}$$

If we want this probability to be at most $1/2f$, we get that

$$\epsilon^2 p\lambda < 4 \ln n + 2 \ln 18f$$

Recall that we also need to know the probability that the minimum cut samples to too many edges, but from Chernoff bounds we immediately get

$$\Pr[\text{minimum cut samples to } > (1+\delta)p\lambda \text{ edges}] < e^{-\delta^2 p\lambda/4}$$

If we want this probability to be at most $1/2f$, we get that

$$\delta^2 p\lambda < 4 \ln 2f$$

Recall also that what we want is that twice the number of edges sampled from the minimum cut is less than thrice the minimum cut of the sample, so that some tree must 2-respect the minimum cut. Translated into the variables above, this statement reads as

$$2(1+\delta) < 3(1-\epsilon)$$

Solving, we find that if we take

$$\epsilon < \frac{1}{3 + \sqrt{\frac{2\ln 2f}{2\ln n + \ln 18f}}}$$

and

$$p > \frac{4 \ln n + 2 \ln 18f}{\epsilon^2 \lambda}$$

then with probability at least $1 - 1/f$ we will get at least one tree that 2-respects the minimum cut. Note that the 4 in the second Chernoff bound above could be tightened a bit, but it only affects how close ϵ is to $1/3$, so it does not matter much.

It now remains to show that the cuts we ignored really did not matter. For this purpose, order the cut values in increasing order, and denote them $c_1(=\lambda), c_2, c_3, \ldots$. So we have already dealt with $c_1 \ldots c_{9n^2}$, and we are now concerned with the rest. By Lemma 3.6.2, $c_{9n^2} > 3\lambda/2$. Thus

$$\Pr[\text{any of } c_{9n^2} \ldots c_{n^3} \text{ samples to } < (1-\epsilon)p\lambda \text{ edges}] < n^3 e^{-(\frac{1+2\epsilon}{3})^2 3p\lambda/4}$$

Assuming $n > f$, we can assume that $1/4 < \epsilon < 1/3$. Plugging in, we get that the probability above is at most $e^{-15/4 \ln n} = n^{-3.75}$, which is clearly negligible.

We now must deal with the remaining cuts $(c_{n^3} \ldots)$. Using Lemma 3.6.1, we get that $c_{n^{2\alpha}} \geq \alpha\lambda$. Rewriting, $c_k > \frac{\ln k}{2 \ln n}\lambda$.

$$\Pr[k\text{th cut sample to } < (1-\epsilon)p\lambda \text{ edges}] < e^{-9 \ln k/4} = k^{-9/4}$$

Applying the union bound, we now need to consider

$$\sum_{k>n^3} k^{-9/4} < \int_{x=n^3}^{\infty} k^{-9/4} = \frac{4}{5} n^{-15/4}$$

Again, this quantity is clearly negligible. This concludes the reanalysis.

As n goes to infinity, we get $\epsilon = 1/3$ and $p = (36 \log n)/\lambda$. For $f = 1/20$ and $n = 32768$, which corresponds to the larger problems we tested on, we get $\epsilon = .284$ and $p = 663/\mathtt{lambda}$. 663 is many more trees than is reasonable.

3.6.2 Tree Packing

As discussed in the theory section, there are at least two completely different possibilities for packing trees. One approach is to use Gabow's algorithm to pack directed spanning trees; the other is to use the fractional packing algorithm of Plotkin, Shmoys and Tardos (PST) to pack undirected trees.

Gabow's Algorithm Our implementation of Gabow's algorithm is mostly straight after the theory. The only significant heuristic we add is to greedily pack depth-first search trees at the beginning, switching to Gabow's algorithm when we get stuck. Our experience is that this heuristic often finds the majority of the trees. We experimented briefly with reducing ϵ (thus increasing the sample size), so that we could stop packing trees when we only had most of them, but we found that the increase in the number of trees caused by the decrease in ϵ negated any benefit of terminating Gabow's algorithm early.

We also considered using the trees found by scan-first search as a heuristic, but we found them unsuitable. For one thing, scan-first search finds undirected trees, so it is typically a factor of two away from the optimum packing. Further, scan-first search is breadth-first in nature, and therefore tends to put many of one node's edges in one tree, thus disconnecting the node unnecessarily, and immediately forcing a switch to Gabow's algorithm. We also note that the fancy way to check for 2-respecting cuts prefers "stringy" trees, such as depth-first search trees. Scan-first search trees may still be a good starting point for PST though.

If we use the theoretically justifiable sampling probability, tree packing seems to be the bottleneck. Unfortunately, the biggest problem we had was that Gabow's algorithm seems to need an explicit representation of all of the trees, so as problem size increases we quickly run out of memory. There are tricks that can be played to reduce the running time of Gabow's algorithm, such as the divide and conquer variant proposed by Karger [34], but it seems that for Gabow's algorithm to be practical, we need to either find a way to implicitly represent the trees, or we need to tighten the analysis of Karger's algorithm so that we do not need to pack so many. As already mentioned, we ended up handling this problem by violating the analysis and declaring the implementation heuristic.

Note that using Gabow's algorithm to pack trees has the advantage that on integer capacity graphs with small minimum cut we can forget about random sampling and 2-respecting cuts and just use Gabow's algorithm to find the minimum cut.

PST Algorithm We have done preliminary experiments with PST, but they are inconclusive. Previous implementation work using PST to find multicommodity flows [44] found that heuristic changes to the algorithm were crucial to good performance. We do not feel that we have worked enough with PST yet to include results on it in this study. It would definitely be interesting to know how it performs. For reference, important issues appear to be selection of a starting point and on-line heuristic adjustment of parameters.

It is easy to implicitly represent the trees in PST, so if the analysis of K cannot be tightened, we suspect that an implementation of K that respects the analysis will have to use PST.

Another major hope for PST is that it will in fact typically pack enough trees that we will only need to look for 1-respecting cuts. Saving the computation of 2-respecting cuts would improve practical performance a great deal.

So we regret that we have not included PST in this study. As far as future implementation work on K goes, we consider PST to be deserving of the highest priority.

3.6.3 Checking for 2-respecting cuts

We implemented both the simple and the fancy methods for checking 2-respecting cuts. We conjectured for a long time that dynamic trees would prove too complicated to be valuable, but it turns out that this is not the case.

The Simple Way The simple method was implemented largely as the theory suggested. We combined the n computations of $f_v^{\downarrow}(w)$ into one tree traversal, and the n computations $g_w^{\downarrow}(v)$ into another. Another change was that we used an explicit test in the middle of the computation to handle the two cases (comparable, incomparable) instead of a fix at the end, as proposed by the theory. It is not clear that this change makes any difference. We do not bother to use a linear time algorithm for computing least common ancestors; rather we use the path compression algorithm of Tarjan [56], which is wonderfully simple and runs in $O(m\alpha(m,n))$ time, where $\alpha(m,n)$ is a functional inverse of Ackermann's function.

A real problem with the method is that this simplest approach to it requires a table of size $O(n^2)$ to keep the values of all the cuts we are interested in as we computed them. This is clearly the simplest thing to do, but use of $O(n^2)$ space incurs a big penalty on sparse graphs. It would be interesting to find another way that is equally simple but space efficient. We eventually decided to select between the simple method and the fancy method on-line, based on the density of the graph.

The Fancy Way We used an implementation of dynamic trees written by Tamas Badics [4] for the first DIMACS implementation challenge. This implementation uses splay trees, which is likely the most practical approach.

We made some non-obvious changes from the theory description in implementing this method. These are not deeply significant, but we give them here for the sake of anyone who wants to implement K himself. Some readers will want to skip on to the experiments chapter.

The theory suggests separating computations for comparable v and w from those for incomparable v and w. The problem is that when looking for an incomparable partner we wish to add $-2c(v,u)$ to neighbors u, and we want to find a w that is incomparable when we do the MinPath operation. In theory, this suggests doing an $\mathsf{AddPath}(v,\infty)$ first, so that no ancestor of v could possibly be the minimum. When we are looking for a comparable partner we wish to add $+2c(v,u)$ for all neighbors u, and obviously we do want an ancestor as the answer. We resolve these problems in our implementation. First, for all edges we compute and store the least common ancestor (LCA) of the endpoints. (Recall that we needed to compute them anyway to find 1-respecting cuts.) Now instead of doing an $\mathsf{AddPath}(u,-2c(v,u))$ for the incomparable case and

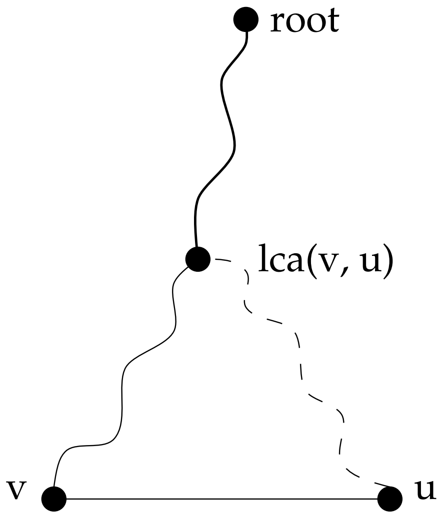

Figure 3.3: Adding $c(v,u)$ up the tree. Along the dashed path we wish to add $-2c(v,u)$, and along the thick path we wish to add $2c(v,u)$. So we add $-2c(v,u)$ from u to the root, and we add $4c(v,u)$ from $\mathsf{LCA}(v,u)$ to the root.

separately an $\mathsf{AddPath}(u, 2c(v,u))$ for the comparable case, we do an $\mathsf{AddPath}(u, -2c(v,u))$ and an $\mathsf{AddPath}(\mathsf{LCA}(u,v), 4c(v,u))$. This puts the right values in the right places (see Figure 3.3). Further, since every node's value is initialized with the value of the cut if its parent edge is cut, and we check 1-respecting cuts first, we only get a comparable w when looking for an incomparable one if there is not one that gives a better cut.

```
Find2RespectingCuts(T)
    initialize a dynamic tree T′ that represents T, and has val(v) = C(v↓)
    while T′ has more than one node
        ProcessBoughs(root of T′)
```

```
ProcessBoughs(v)
    if v has multiple children
        for each child w of v
            (partner, value) = ProcessBoughs(w)
            if (partner, value) is not NIL (w is top of bough)
                for all edges {v, u} (undo dynamic tree ops)
                    AddPath(u, 2c(v, u))
                    AddPath(LCA(u, v), −4c(v, u))
                contract(v, w)
        return NIL
    if v has one child w
        (partner, value) = ProcessBoughs(w)
        if (partner, value) is NIL, return NIL (v not on a bough)
        (partner′, value′) = ProcessNode(v)
        contract(v, w)
        if value′ < value, return (partner′, value′)
        else return (partner, value)
    else (v is a leaf)
        return ProcessNode(v)
```

```
ProcessNode(v)
    for all edges {v, u}
        AddPath(u, −2c(v, u))
        AddPath(LCA(u, v), 4c(v, u))
    for all edges {v, u}
        x = MinPath(u)
        if val(x) + C(v↓) < λ̂
            λ̂ = val(x) + C(v↓)
            partner = x
    x = MinPath(u)
    if val(x) − C(v↓) < λ̂
        λ̂ = val(x) + C(v↓)
    return (partner, λ̂)
```

Chapter 4

Experiments

In this chapter we discuss the experiments we carried out on the implementations described in the last chapter. We begin by describing the design of our experiments. We then discuss the results.

4.1 Experiment Design

The most important part of running experiments is the inputs that are tested. It is of course impossible to try everything; subjective choices were necessarily made in the design of our experiments. In this section we describe and justify those choices. We begin by laying out our goals, which guided these decisions. We then describe the families of inputs we chose, and give details on precisely which experiments we ran.

4.1.1 Goals

As stated in the introduction, our goal in this study is to obtain efficient implementations of all the algorithms and meaningful comparisons of their performance. Of course, it is not obvious how to define meaningful in this context. As an approximate definition, we adopt the following rules:

1. running times on real-world problems are meaningful

2. comparisons to previous work are meaningful

3. running times on problems that expose weaknesses in the algorithms and/or implementations are meaningful

4. running times that differ by a small constant factor are not meaningful

The justification for rule 1 is obvious. Performance on real-world problems is a direct measure of real-world performance. It is also clear that rule 2 is reasonable, as our results would be questionable if they differed too dramatically from previous work. Rule 3 may seem objectionable, in that the kinds of graphs that expose weaknesses may never come up in applications, but we maintain that it is important to know just how bad performance is in the worst case. For example, if one implementation typically wins by a factor of five, but has a bad case where it loses by a factor of 10, one would probably be happy to use it and hope the bad case does not happen. However, if the bad case causes the implementation to lose by a factor of 10,000, one might be more hesitant about blindly hoping that the bad case does not occur.

We justify rule 4 based on the fact that small factors come and go easily. In all likelihood, a good (and determined) programmer could speed up all of our implementations by a factor of two just by carefully optimizing the source code. Likewise, machine dependencies, such as cache size, are liable to have small effects that we might be able to fix, but our fixes might be unnecessary or even undesirable on another machine. We are not interested in such details. We hope to be able to recommend an algorithm to use and provide a starting implementation, but someone who is interested in the absolute best performance will have to (and probably want to) do the final optimization himself.

Note that as a corollary of rule 4, we chose to look for minimum cut values, not the actual cuts. This decision simplifies the code a bit, and ensures that the implementations do not take a long time simply because they find many cuts. (Any of the algorithms can discover $\Omega(n)$ cuts; assuming it takes $\Omega(n)$ time to save a cut when found, the time spent saving cuts could be $\Omega(n^2)$, which might dominate the runtime.) It is easy to adapt our codes to actually find the minimum cut without affecting the running time by more than a factor of two: first find the minimum cut value, then run the algorithm again and stop when we first find a cut with the same value, outputting this cut. Since this modification can change the running time by at most a factor of two, we deemed it unnecessary to worry about it in our tests.

4.1.2 Problem Families

We chose several different families of inputs to cover the different types of tests we decided were meaningful.

Class name	**Brief description**
TSP	TSP instances
PRETSP	Preprocessed TSP instances
NOI1–NOI6	Random graphs with "heavy" components (after NOI)
REG1–REG2	Regular random graphs
IRREG	Irregular random graphs
BIKEWHE	Bicycle wheel graphs
DBLCYC	Two interleaved cycles
PR1–PR8	Two components with a min-cut between them (after PR)

Table 4.1: *Summary of problem families.*

Subproblems from a Traveling Salesman Problem Solver

A state of the art method for solving Traveling Salesman Problem (TSP) instances exactly uses the technique of *cutting planes*. The set of feasible traveling salesman tours in a graph induces a convex polytope in a high-dimensional vector space. Cutting plan algorithms find the optimum tour by repeatedly solving a linear programming relaxation of an integer programming formulation of the TSP and adding linear inequalities that cut off undesirable parts of the polytope until the optimum solution to the relaxed problem is integral. One set of inequalities that has been very useful is *subtour elimination constraints*, first introduced by Dantzig, Fulkerson, and Johnson [15]. The problem of identifying a subtour elimination constraint can be rephrased as the problem of finding a minimum cut in a graph with real-valued edge weights. Thus, cutting plane algorithms for the traveling salesman problem must solve a large number of minimum cut problems (see [43] for a survey of the area). We obtained some of the minimum cut instances that were solved by Applegate and Cook [3] in their TSP solver. These are clearly desirable test data, as they are from a "real-world" application.

The Padberg-Rinaldi heuristics are very effective on the TSP instances. In order to factor out the time spent in preprocessing, for each TSP instance we made a smaller instance by running PR passes until some pass fails to do any contractions. (Note that running PR passes until one pass fails is not the same as exhaustively applying the PR tests.) We refer to these reduced instances as PRETSP. We tested the implementation on both the original TSP problems and the PRETSP problems.

Table 4.2 gives a summary of these instances, including their "names" which correspond to the original TSP problems.

Note that these problems are smaller than we would have liked. Several PRETSP instances have only two nodes, and the largest PRETSP instance has only 607 nodes. The running times of the best algorithms are therefore small, making it hard for us to really distinguish them. We were unable to obtain larger instances; remember that finding a minimum cut is only a subroutine of a TSP solver, and the whole algorithm apparently takes too long for TSP researchers to be running on much larger graphs.

Random Graphs with "Heavy" Components

A natural type of graph on which to run a minimum cut algorithm is the type that is always drawn to exhibit the problem: two well connected components connected by low capacity edges. Nagamochi et al. [48] used a family of graphs that generalizes this idea. We use the same family, which is parameterized as follows:

n the number of vertices in the graph

d the density of edges as a percent (*i.e.* $m = \frac{d}{100}\frac{n(n-1)}{2}$)

k the number of "heavy" (well-connected) components

P the scale between intercomponent and intracomponent edges

Problem number	Problem name	n	m	n'	m'
1	tsp.att532.x.1	532	787	20	38
2	tsp.vm1084.x.1	1084	1252	19	36
3	tsp.vm1748.x.1	1748	2336	77	131
4	tsp.d1291.x.1	1291	1942	88	185
5	tsp.fl1400.x.1	1400	2231	148	300
6	tsp.rl1323.x.1	1323	2169	113	221
7	tsp.rl1323.x.2	1323	2195	106	208
8	tsp.r15934.x.1	5934	7287	150	292
9	tsp.r15934.x.2	5934	7627	261	517
10	d15112.xo.19057	15112	19057	605	1162
11	pla33810.xo.38600	33810	38600	2	1
12	pla33810.xo.39367	33810	39367	2	1
13	pla33810.xo.39456	33810	39456	2	1
14	pla85900.xo.102596	85900	102596	2	1
15	pla85900.xo.102934	85900	102934	2	1
16	pla85900.xo.102988	85900	102988	52	90
17	usa13509.xo.15631	13509	15631	325	561
18	usa13509.xo.17048	13509	17048	477	920
19	usa13509.xo.17079	13509	17079	449	861
20	usa13509.xo.17111	13509	17111	454	886
21	usa13509.xo.17130	13509	17130	403	786
22	usa13509.xo.17156	13509	17156	532	1029
23	usa13509.xo.17156a	13509	17156	501	950
24	usa13509.xo.17183	13509	17183	492	946
25	usa13509.xo.17193	13509	17193	549	1072
26	usa13509.xo.17210	13509	17210	465	926
27	usa13509.xo.17303	13509	17303	542	1064
28	usa13509.xo.17358	13509	17358	573	1104
29	usa13509.xo.17375	13509	17375	476	945
30	usa13509.xo.17386	13509	17386	607	1150
31	usa13509.xo.17390	13509	17390	557	1091
32	usa13509.xo.17494	13509	17494	505	971

Table 4.2: Summary of TSP and PRETSP instances. n and n' is the number of nodes in the TSP and PRETSP instances, respectively. m and m' are the corresponding numbers of edges.

The graph is constructed by first taking n vertices and randomly coloring them with k colors. We then add one random cycle on all vertices, so the graph will be connected, and add the remaining $m-n$ edges at random. Every edge added gets a random capacity. If the endpoints have different colors, the capacity is chosen uniformly at random from $[1, 100]$; otherwise the capacity is chosen uniformly at random from $[1, 100P]$.

Following [48], we tested on 6 subfamilies. Our families are the same in spirit as those of [48], but we use larger problem sizes and we added some data points where we felt it was appropriate.

Family	n	d	k	P
NOI1	300,400,500,600 700,800,900,1000	50	1	300,400,500,600, 700,800,900,1000
NOI2	300,400,500,600 700,800,900,1000	50	2	300,400,500,600, 700,800,900,1000
NOI3	1000	5,10,25,50,75,100	1	1000
NOI4	1000	5,10,25,50,75,100	2	1000
NOI5	1000	50	1,2,3,5,7,10,20,30,33,35 40,50,100,200,300,400,500	1000
NOI6	1000	50	2	5000,2000,1000,500 250,100,50,10,1

Families NOI1 and NOI2 study the effect of varying the number of vertices. Families NOI3 and NOI4 study the effect of varying the density of the graph. Family NOI5 studies the effect of varying the number of components, and family NOI6 studies the effect of varying the ratio between the weights of the intercomponent and intracomponent edges.

Regular Random Graphs

Recall that in the analysis of the original KS, we lower bound m with $\lambda n/2$. It follows that a graph where this bound is tight is liable to be an interesting graph for KS. If uncapacitated, such a graph is interesting in general, because it has the minimum number of edges possible given its minimum cut value. (An uncapacitated graph must have λ edges incident to every vertex, since otherwise some vertex defines a smaller cut.) Recall also that one of the heuristics in NI computes lower bounds on cut values between edge endpoints, and can lead to many contractions in one phase. It makes sense that a graph that has as few edges as possible might cause this heuristic to fail.

We achieve this extreme case with a λ-regular graph. In particular, we take the union of $\lambda/2$ random cycles. Preliminary experiments with a union of λ random matchings gave similar results.

Family	n	λ
REG1	1000, 2000, 4000, 8000, 16000 1000	8 16 32 64 128 256 512 1024
REG2	128, 256, 512, 1024, 2048	$n/8$

REG1 tests the effect of varying n and λ on sparse graphs. REG2 tests the effect of varying n on dense graphs.

Irregular Random Graphs

An obvious question about the previous family is what happens if symmetry is broken a little bit. In particular, we consider taking the union of λ random matchings (or cycles), and then adding some edges of another random matching. It is not obvious that this family will produce different results, but it turns out that NI changes behavior in interesting ways. We add another parameter e, the number of extra edges, to the parameters from the REG families.

Family	n	λ	e
IRREG	4000	8, 9	0, 2, 4, 16, 64, 256, 1024, 2000, 2976, 3744, 3936, 3984, 3996, 3998, 4000

Bicycle Wheels

Another extreme graph is a cycle. An uncapacitated cycle has $\binom{n}{2}$ minimum cuts, a value that matches the upper bound, and $n\lambda/2$ edges, a value that matches the lower bound. A cycle also has only one undirected spanning tree, despite having minimum cut value two, so it exhibits the extreme case for the size of a tree packing.

Unfortunately, PR2 applies at every vertex in a cycle, so PR preprocessing always solves cycles. One natural way to try to overcome this problem is to make a "wagon wheel" instead. That is, add an extra vertex that is connected to every vertex on the cycle. If the capacities of the new edges are small compared to the cycle edges, then the graph is still very much like a cycle in terms of its cuts, but PR2 no long applies. Unfortunately, now PR3 applies at every vertex. So we go one step further: we take a cycle and add two extra vertices, one connected to every other node, and the other connected to the remaining nodes. We also connect the two added vertices. We refer to this graph as a bicycle wheel, as that is precisely what it looks like. (See Figure 4.1.2). Note that this graph is now immune to all the PR tests. (In fact, it is the example of a PR immune graph we gave in Section 2.2.1.)

We pick the capacities so that all trivial cuts have the same value. This choice causes the "rim" to have large capacity, and the "spokes" to have small capacity, which means that the cuts are still very much like those of a cycle. The only parameter then is the number of vertices, so that is all we vary.

Family	n
BIKEWHE	1024, 2048, 4096, 8192, 16384, 32768

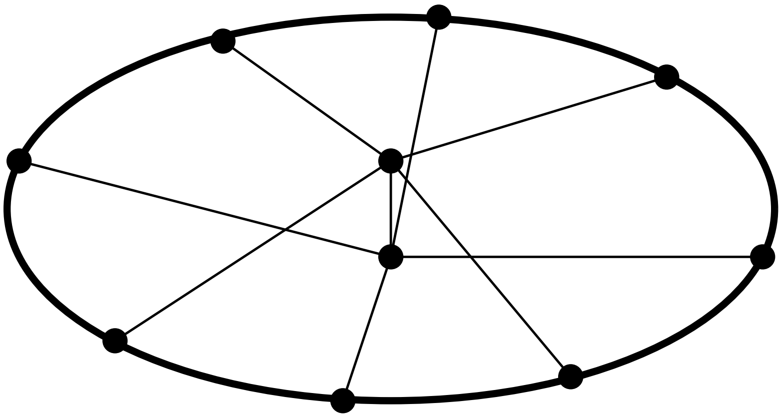

Figure 4.1: A 10-vertex bicycle wheel graph.

Two Interleaved Cycles

Another way to get a graph that is basically a cycle, but is immune to the PR tests, is to use two cycles. For this family we use an n-node cycle with capacity 1000, and we make a second cycle by connecting every third node of the original cycle with a unit capacity edge.

In order to make this family a little more interesting, we also "hide" a minimum cut in the middle. That is, we take two opposite cycle arcs and decrease their capacity by three. We then increase the capacity of four of the second cycles' edges by three, such that all trivial cuts still have value 2002. In the process, however, we have created a cut of value 2000. Note there are also $\Omega(n^2)$ cuts of value 2006. This modification is cumbersome to describe in words, but the picture is clear. See Figure 4.2.

Family	n
DBLCYC	1024, 2048, 4096, 8192, 16384, 32768

PR

One final problem family is one used by Padberg and Rinaldi [51]. Our only use of this family is to check the effectiveness of our PR strategies against those of Padberg and Rinaldi.

This family includes two different types of graphs. The first type is a random graph with an expected density d. The second type is a random graph that consists of two components connected by "heavy" edges, with "light" edges going between the components, thus the minimum cut is very likely to separate the two components (similar to the NOI families). The generator takes three parameters

- n - the number of vertices,
- d - the density (as a percentage),

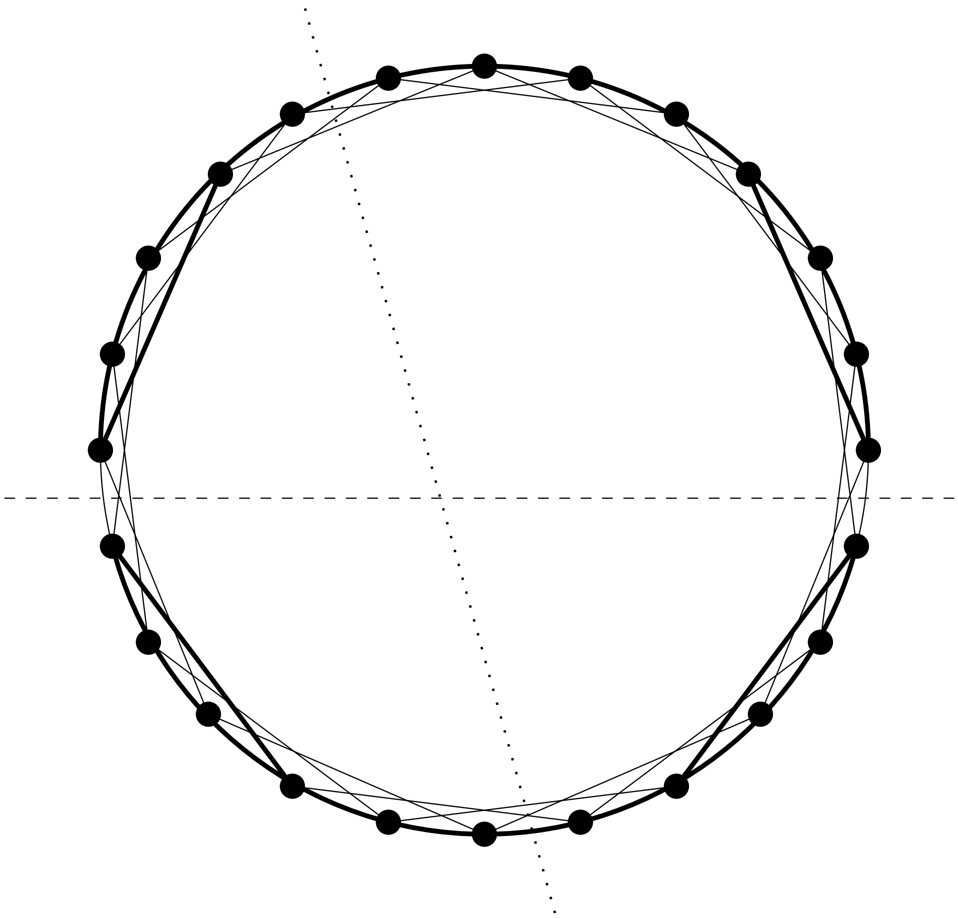

Figure 4.2: Two interleaved cycles. The outer cycle edges have capacity 1000, except for the two thin ones, have which capacity 997. The inner edges have capacity 1, except for the 4 thick ones, which have capacity 4. The dashed line shows the minimum cut (value 2000). The dotted line shows one of many near minimum cuts (value 2006).

- c - the type of graph to generate (1 or 2).

If $c = 1$, for each pair of vertices, with probability d, we include an edge with weight uniformly distributed in $[1, 100]$. If $c = 2$, we split the graph into two components, one containing vertices 1 through $n/2$ and the other containing vertices $n/2 + 1$ through n. Again, for each pair of vertices, we include an edge with probability d. If the two vertices are in the same component, the edge weight is chosen uniformly from $[1, 100n]$, but if the vertices are in different components, the edge weight is chosen uniformly from $[1, 100]$.

Family	n	d	c
PR1	100,200,300,400	2	1
PR2	100,200,300,400	10	1
PR3	100,200,300,400	50	1
PR4	100,200,300,400	100	1
PR5	100,200,300,400,500,1000,1500,2000	2	2
PR6	100,200,300,400	10	2
PR7	100,200,300,400	50	2
PR8	100,200,300,400	100	2

As we wish to compare directly to Padberg and Rinaldi, these values are precisely those used by Padberg and Rinaldi in their paper [51].

4.1.3 Codes

As discussed in the previous chapter, for each algorithm we made numerous decisions about the implementation. While the process of implementation involved testing many of these decisions, there are far too many for us to attempt to present data on everything we tried. We picked several important variations on which to report data. See Table 4.1.3 for a summary.

Code	Description
`ho`	HO with all heuristics
`ho_nopr`	HO without PR heuristics
`ho_noxs`	HO without excess detection
`ho_noprxs`	HO without PR heuristics or excess detection
`hybrid`	Nagamochi *et al* implementation of NI
`ni`	NI with all heuristics
`ni_nopr`	NI without PR heuristics
`ks`	KS with all heuristics
`ks_nopr`	KS without internal PR heuristics
`k`	K with all heuristics

Table 4.3: Summary of the implementations we tested

Notice that the suffix `_nopr` does not mean the same thing in all cases. For HO and NI we show what happens when PR heuristics are disabled entirely, whereas for KS we never consider disabling PR preprocessing, and for K we never disable the PR heuristics at all. This decision is based on experience with the PR tests.

It turns out that no one algorithm is clearly best, and we did attempt to make a hybrid algorithm that would fill this role. We can give a good idea of what such an implementation would look like based on the data we got from the implementations above.

4.1.4 Setup

Our experiments were conducted on a machine with a 200MHz PentiumPro processor, 128M of RAM, and a 256K cache. Our codes are written in C and compiled with the GNU C compiler (`gcc`) using the `O4` optimization option. All of the Monte Carlo implementations (`ks`, `ks_nopr`, `k`) used 95% as a minimum success probability.

We averaged five runs wherever randomness was involved. That is, for the problem families that are constructed randomly, we constructed five instances for each setting of the parameters. Further, for the randomized algorithms, we did five runs on each instance. We report averages.

As mentioned in Section 4.1.1, our implementations do not actually output the minimum cut, or save the minimum cut in any special data structure. However, at the time the minimum cut is encountered they do have the minimum cut stored in some internal data structure from which it could easily be extracted.

In order to see why different implementations had different performances, we recorded many quantities in addition to total running time:

For all implementations we measured:

total running time not including time to input the graph

discovery time the time at which the algorithm first encountered the minimum cut. This quantity tells us two things. First, if we use the two pass method to get the actual cut, as described in Section 4.1.1, the discovery time will be the running time of the second pass. So running time plus discovery time should be the time to find and output a minimum cut. Second, for KS, discovery times tells us how many iterations of the recursive contraction algorithm we actually needed to run. If discovery times are always far less than running times, we might suspect that the analysis is not tight.

edge scans the number of times an edge was examined. Examining edges is a basic unit of work that all the algorithms perform. Hence this quantity provides some sort of machine independent measure of running time. This is a basic unit of work that is common to all of the codes.

For implementations that perform PR tests we measured:

preprocessing time the time spent preprocessing the graph with PR tests.

initial PR contractions the number of contractions done by the PR preprocessing.

internal PR contractions the number of contractions due to PR tests while running the main algorithm.

For HO implementations we measured:

s-t cuts the number of s-t cut (max-flow) computations, not counting single node layers.

average problem size the average number of vertices in an s-t cut problem.

one node layers the number of times a sleeping layer has exactly one node. (Recall that we process this case specially.)

excess contractions the number of contractions due to the excess detection heuristic.

For NI implementations we measured:

phases the number of scan-first searches executed.

For KS implementations we measured:

leaves the number of leaves of the recursion tree.

For K implementations we measured:

packing time the amount of time spent packing trees.

respect time the amount of time spent checking for 1 and 2-respecting cuts.

4.2 Results

In this section we discuss our results. The overall result is that `ho` and `ni` are best, although each has a bad case, and on bicycle wheels they both lose asymptotically to `k` and `ks`. We give more details by first discussing the results on each problem family, and then discussing each algorithm.

In this section we present most of the data in the form of plots. Full data appears in tabular form in the Appendix. The plots always have $\log(\text{running time})$ as the vertical axis. For families where we are varying the size or density of the graph, we also use a logarithmic scale for the horizontal axis. Since we expect the running time of the algorithms to be expressible as $c_1 n^{c_2}$, log-log plots are appropriate: the y-intercept tells us c_1 and the slope tells us c_2. So parallel lines correspond to algorithms with the same asymptotic performance and different constant factors, and different slopes correspond to different asymptotic performance. Where appropriate we use a linear regression to compute the slopes and intercept of the best fit line and report these performance functions in a table.

Note that our timer was not precise below 0.01 seconds, and we cannot plot 0.00 on a log scale, so any timer results of 0.00 were translated to 0.01 for plotting purposes. Such small values probably should not be trusted in any case.

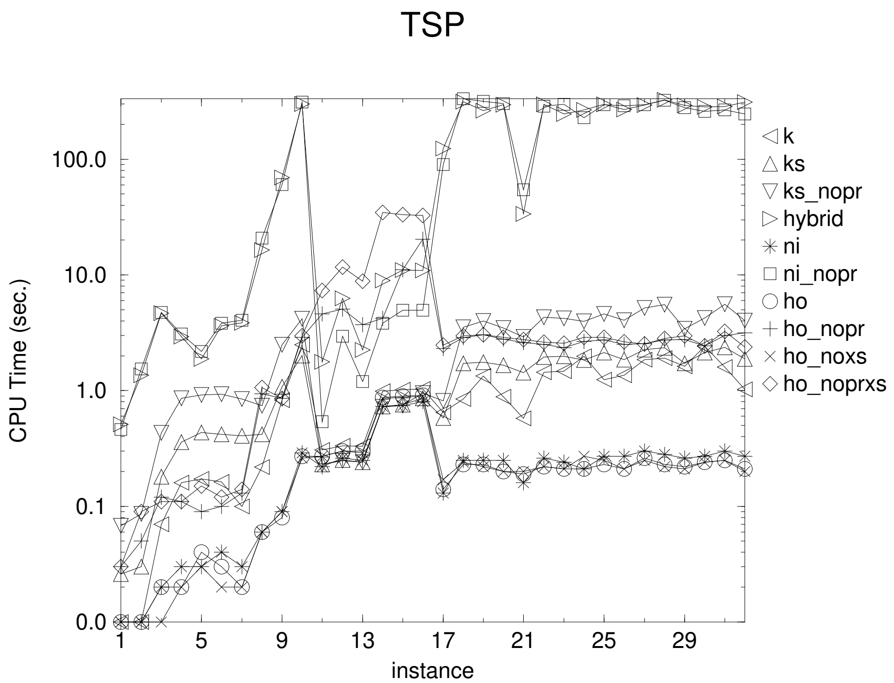

Figure 4.3: All implementations on the TSP instances. Note that the x-axis of this plot has no meaning; the points are connected by lines because the lines seem to make it easier to read the plot.

4.2.1 Results by Problem Family

TSP

The TSP family turned out to be a study in the effectiveness and subtlety of PR tests. The most striking result here is the difference between `hybrid` and `ni`, which for the "USA" instances (17–32) is approximately a factor of 1000 (see Figure 4.3) . This difference is almost entirely due to the PR strategy. Recall that `hybrid` does have PR tests, yet it behaves like `ni_nopr`. Most of the difference is our PR preprocessing, which reduces the size of the USA problems by a factor of 10 to 15 in about a tenth of a second. However, if we factor out this difference by preprocessing the instances before running the codes on them (Figure 4.4), we find that `ni` is still gaining something over both `ni_nopr` and `hybrid`, so our internal PR strategy is also gaining us something.

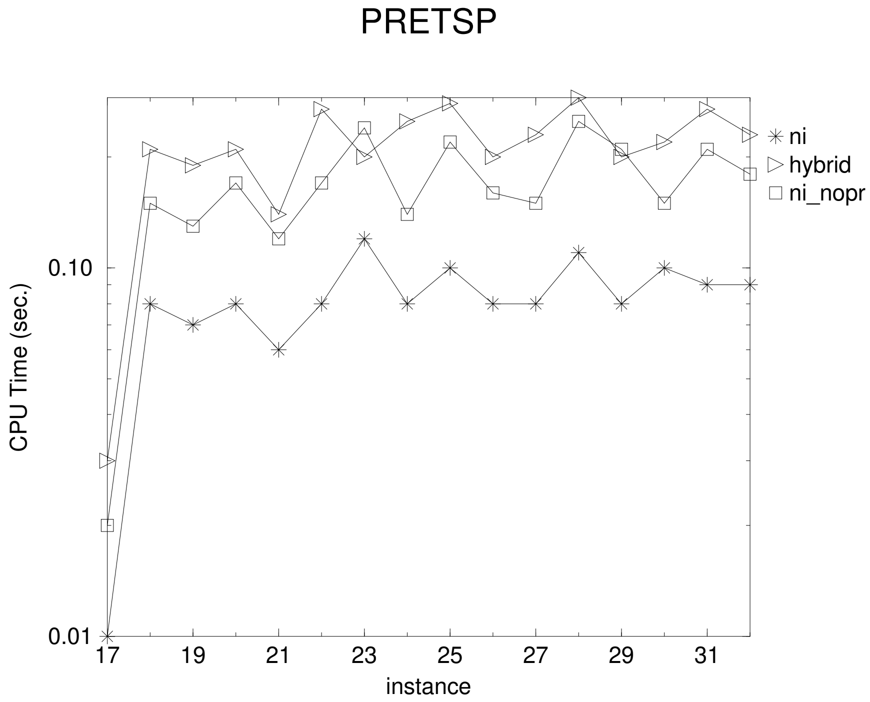

Figure 4.4: NI variants on preprocessed TSP USA instances (PRETSP 17–32).

NOI

Overall, random graphs with heavy components serve as a demonstration of the good case for NI. These graphs have many "extra" edges, and as one would hope, all of the implementations of NI were able to exploit this property to run in near-linear time. Further, using Matula's approximation algorithm to get a good cut upper bound and computing a sparse certificate based on that value was sufficient to solve almost every instance, so the behavior of `k` on almost all of these problems is the behavior of this preprocessing step. Notice that `k` is still several (roughly 4)

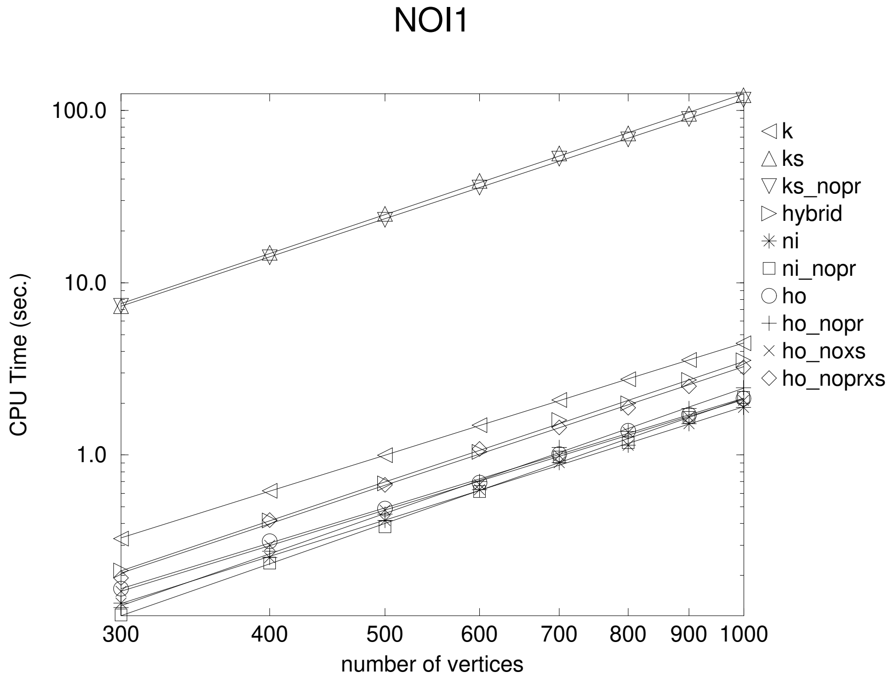

Figure 4.5: All implementations on random graphs (varying size).

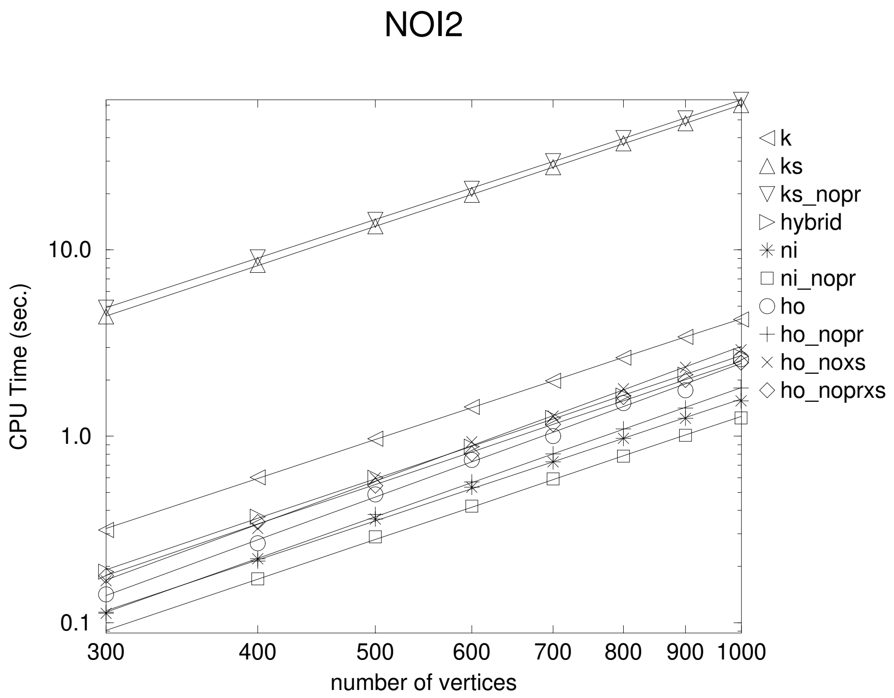

Figure 4.6: All implementations on random graphs with 2 heavy components (varying size).

times slower than `ni`. There are two reasons for this. First, since the contractions done by Matula's algorithm must be undone, `k` has extra overhead in the contraction code to allow contractions to be undone. Second, `k` must always do at least two sparse certificate computations: at least one for Matula's algorithm and then the one that uses the value computed by Matula's algorithm. `ni`, on the other hand, typically needs only one sparse certificate computation for these graphs.

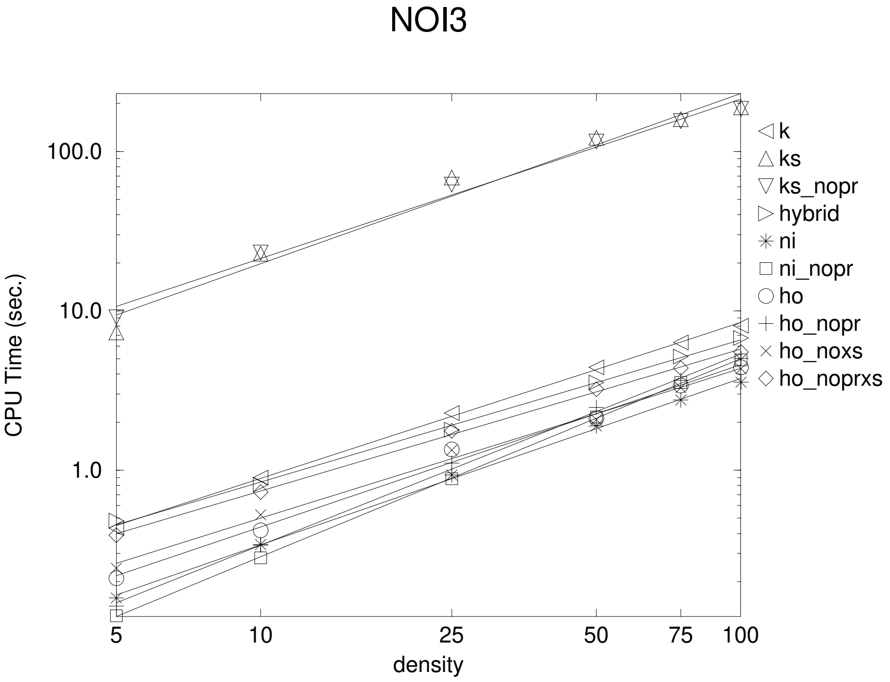

Figure 4.7: All implementations on random graphs (varying density).

Varying the number of nodes and the density of the graphs, all the implementations behave with similar asymptotics (Table 4.4 and Figures 4.5–4.8). In fact, only KS distinguishes itself, and that is for having significantly worse constant factors. Actually, KS's analysis is failing it on these problems. The variant was designed with graphs of this nature in mind, and indeed `ks` typically ends up with very shallow recursion trees. However, while it appears that the success probability here should be constant, we could not find the right way to determine that fact on-line. The problem is that we cannot just look at the depth of the trees and use the actual depth to revise our estimate of success probability, because if we condition on the fact that a recursion tree has small depth, we find that the probability we have contracted a minimum cut edge increases. So all we can say is that on these families `ks` actually performs quite well, but we do not know how to recognize this fact on-line and terminate early.

Varying the number of components is more interesting (Figure 4.9). For one thing, there is a threshold value, after which the PR preprocessing solves the problem. This threshold appears to occur when the intercomponent cuts become as large as the the intracomponent cuts. Note that we can see the slowdown `k` has in the contraction code, because after we cross the threshold it runs a constant factor slower than the other implementations with PR preprocessing. We also see that excess detection can sometimes fill in for PR tests, as `ho_nopr` improves performance at the

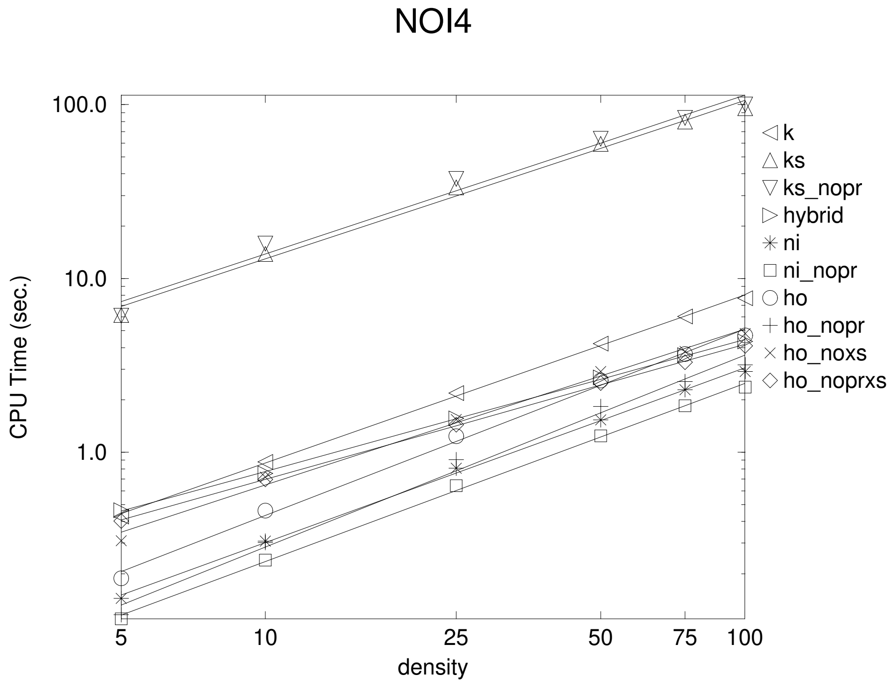

Figure 4.8: All implementations on random graphs with 2 heavy components (varying density).

	NOI1	NOI2	NOI3	NOI4
`k`	$1.43 \times 10^{-6} \times n^{2.16}$	$1.46 \times 10^{-6} \times n^{2.16}$	$9.31 \times 10^{-2} \times d^{0.98}$	$9.36 \times 10^{-2} \times d^{0.97}$
`ks`	$1.28 \times 10^{-5} \times n^{2.33}$	$1.86 \times 10^{-5} \times n^{2.17}$	$1.69 \times d^{1.07}$	$1.59 \times d^{0.91}$
`ks_nopr`	$1.60 \times 10^{-5} \times n^{2.29}$	$2.60 \times 10^{-5} \times n^{2.13}$	$2.13 \times d^{1.00}$	$1.69 \times d^{0.91}$
`hybrid`	$3.77 \times 10^{-7} \times n^{2.32}$	$6.97 \times 10^{-7} \times n^{2.20}$	$1.08 \times 10^{-1} \times d^{0.89}$	$1.35 \times 10^{-1} \times d^{0.76}$
`ni`	$5.39 \times 10^{-7} \times n^{2.18}$	$4.88 \times 10^{-7} \times n^{2.17}$	$3.06 \times 10^{-2} \times d^{1.04}$	$2.99 \times 10^{-2} \times d^{1.00}$
`ni_nopr`	$1.24 \times 10^{-7} \times n^{2.41}$	$3.24 \times 10^{-7} \times n^{2.20}$	$1.63 \times 10^{-2} \times d^{1.24}$	$2.22 \times 10^{-2} \times d^{1.03}$
`ho`	$9.75 \times 10^{-7} \times n^{2.11}$	$1.78 \times 10^{-7} \times n^{2.38}$	$4.23 \times 10^{-2} \times d^{1.02}$	$3.67 \times 10^{-2} \times d^{1.07}$
`ho_nopr`	$1.37 \times 10^{-7} \times n^{2.42}$	$2.21 \times 10^{-7} \times n^{2.31}$	$2.14 \times 10^{-2} \times d^{1.20}$	$2.23 \times 10^{-2} \times d^{1.11}$
`ho_noxs`	$8.62 \times 10^{-7} \times n^{2.13}$	$1.94 \times 10^{-7} \times n^{2.40}$	$5.74 \times 10^{-2} \times d^{0.94}$	$8.19 \times 10^{-2} \times d^{0.90}$
`ho_noprxs`	$4.40 \times 10^{-7} \times n^{2.29}$	$6.36 \times 10^{-7} \times n^{2.20}$	$9.53 \times 10^{-2} \times d^{0.89}$	$1.17 \times 10^{-1} \times d^{0.78}$

Table 4.4: Asymptotic behavior of the implementations on graphs with heavy components.

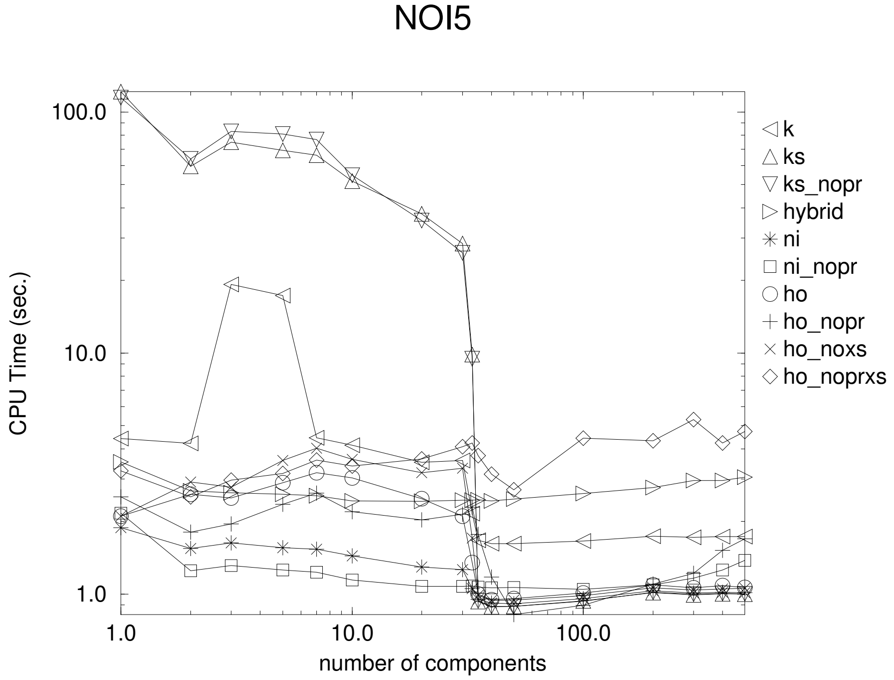

Figure 4.9: All implementations on random graphs with heavy components (varying number of components).

same threshold, whereas ho_noprxs does not change behavior.

Another interesting aspect of varying the number of components is that for five and seven components, preprocessing with Matula's algorithm and one sparse certificate computation does not solve the problem. For all other numbers of components it does. Notice that in the two cases where it has to do some work, k reveals that its performance on these graphs is better than ks, but not very good.

Finally, varying the capacity of the intracomponent edges (Figure 4.10), we see that for very high values, most of the implementations improve performance. They all seem to improve at the same point, but for very different reasons. For ks, the improvement comes because the trees become very shallow when there is so much extra edge capacity to be picked for contraction. For ni, fewer sparse certificate computations are necessary, as it finds more excess edges to contract. For ho, excess detection quickly causes most of the vertices to be contracted away.

REG

Regular random graphs (Tables 4.5 and 4.6, Figures 4.11–4.13) are the bad case for NI. In fact, they induce NI's worst case $\Omega(mn)$ time performance. This fact is immediately apparent on both sparse and dense graphs. The only place where any NI implementation manages to perform well is when the graphs are very dense, and the PR tests kick in.

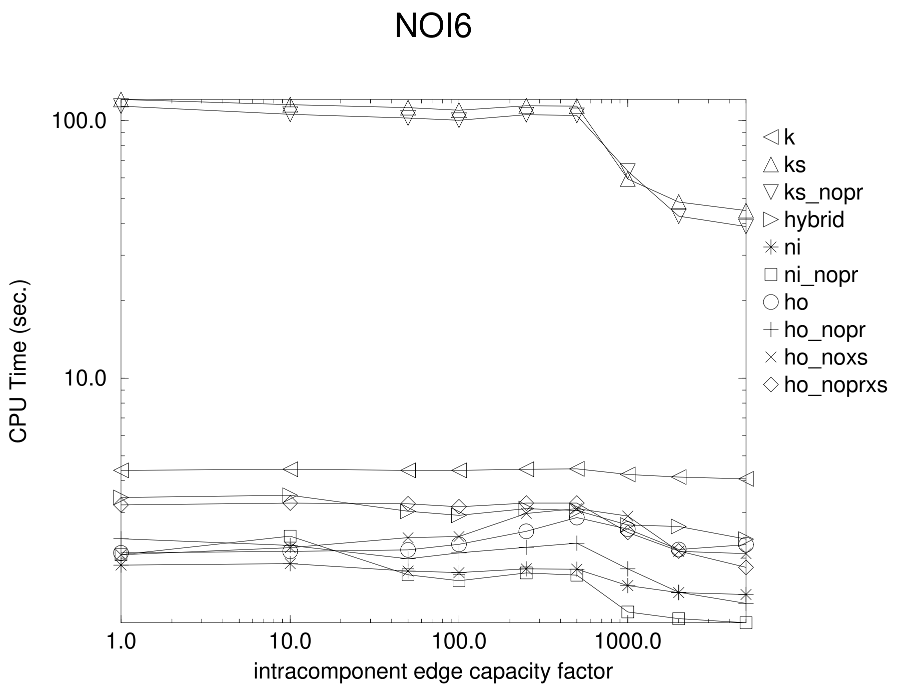

Figure 4.10: All implementations on random graphs with heavy components (varying "heaviness" of components).

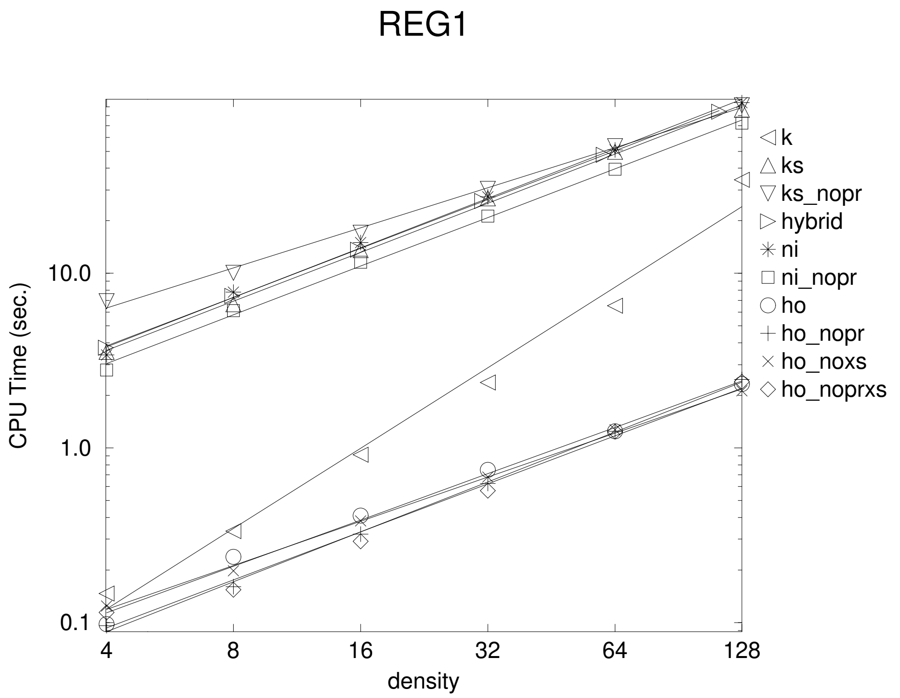

Figure 4.11: All implementations on sparse regular random graphs (varying density).

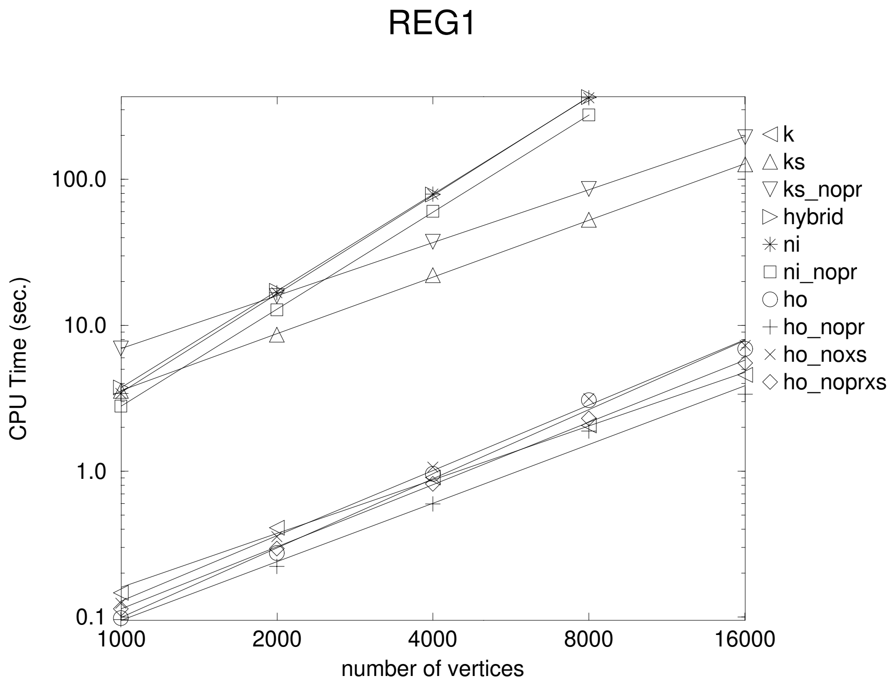

Figure 4.12: All implementations on sparse regular random graphs (varying size).

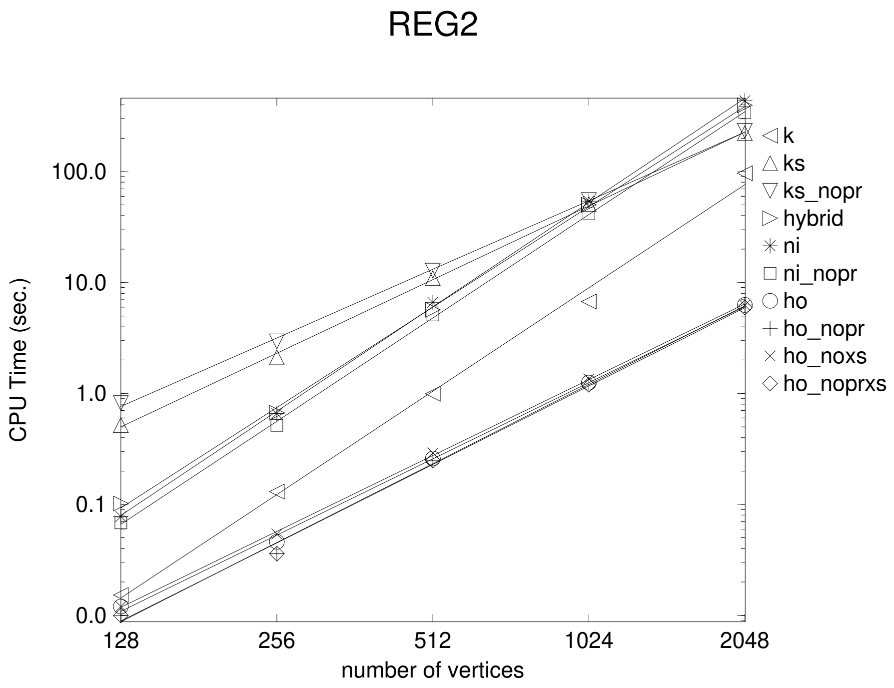

Figure 4.13: All implementations on dense regular random graphs.

	REG1
`k`	$8.10 \times 10^{-6} \times n^{1.13} d^{1.42}$
`ks`	$1.80 \times 10^{-4} \times n^{1.25} d^{0.93}$
`ks_nopr`	$5.92 \times 10^{-4} \times n^{1.19} d^{0.77}$
`hybrid`	$3.55 \times 10^{-7} \times n^{2.17} d^{0.90}$
`ni`	$3.02 \times 10^{-7} \times n^{2.18} d^{0.92}$
`ni_nopr`	$2.44 \times 10^{-7} \times n^{2.18} d^{0.92}$
`ho`	$1.09 \times 10^{-6} \times n^{1.52} d^{0.83}$
`ho_nopr`	$1.31 \times 10^{-6} \times n^{1.42} d^{0.94}$
`ho_noxs`	$2.84 \times 10^{-6} \times n^{1.39} d^{0.80}$
`ho_noprxs`	$2.66 \times 10^{-6} \times n^{1.35} d^{0.86}$

Table 4.5: Asymptotic behavior of the implementations on sparse regular random graphs. (These fits do not include the unusual cases $n = 1000$, $d = 256, 512$.)

	REG2
`k`	$4.34 \times 10^{-9} \times n^{3.09}$
`ks`	$1.12 \times 10^{-5} \times n^{2.21}$
`ks_nopr`	$3.59 \times 10^{-5} \times n^{2.05}$
`hybrid`	$4.25 \times 10^{-8} \times n^{3.01}$
`ni`	$2.07 \times 10^{-8} \times n^{3.12}$
`ni_nopr`	$2.02 \times 10^{-8} \times n^{3.09}$
`ho`	$1.68 \times 10^{-7} \times n^{2.28}$
`ho_nopr`	$9.03 \times 10^{-8} \times n^{2.37}$
`ho_noxs`	$1.95 \times 10^{-7} \times n^{2.27}$
`ho_noprxs`	$9.92 \times 10^{-8} \times n^{2.35}$

Table 4.6: Asymptotic behavior of the implementations on dense regular random graphs.

These graphs are also sufficiently sparse that `k` just runs Gabow's minimum cut algorithm. Gabow's algorithm is apparently competitive with `ho` when the cut value is very small, but as the value increases performance quickly degrades. The transition away from Gabow's algorithm is the reason for the sudden change in behavior that can be seen in Figure 4.11 when the graph gets dense. We were unable to run large enough problems to compare `k` and `ho` for the case where the graph is sparse, but has at least $\Omega(\log n)$ cycles.

Note that `k`'s terrible asymptotic performance on the dense instances is an artifact of small problems. For these instances, the number of nodes is small enough that even though they have $\Omega(n^2)$ edges, `k` is deciding that they are sparse enough to use Gabow's algorithm on. The $n = 2048$ case does not quite fall on the fit line in Figure 4.13, because it is the first instance where Gabow's algorithm is not used. The asymptotic performance will improve after this point.

IRREG

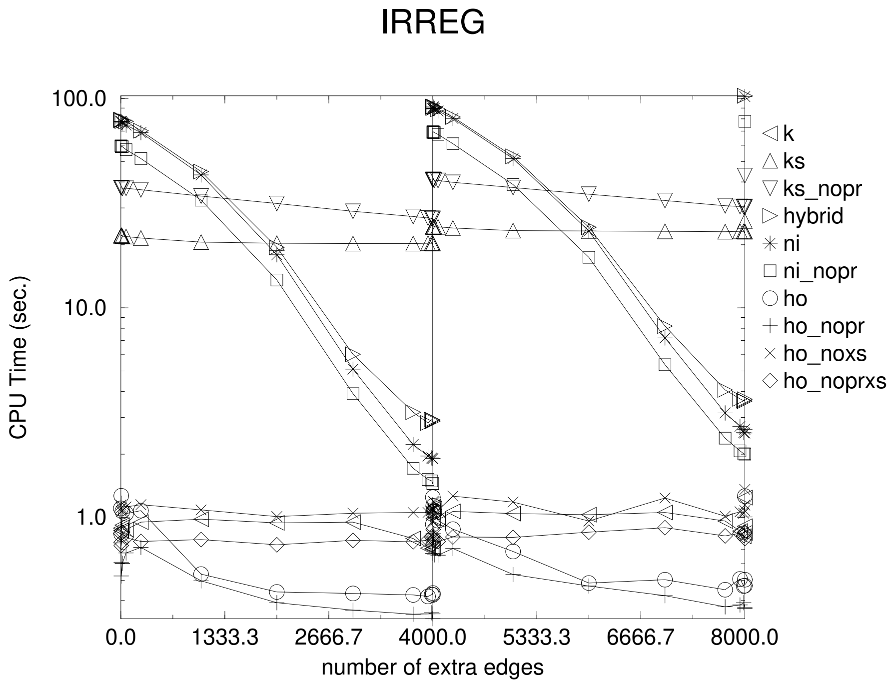

Figure 4.14: All implementations on irregular random graphs.

Irregular random graphs (Figure 4.14) mainly show the significant dependence NI has on graph regularity. The performance of all implementations of NI tend to do poorly when the input is very regular, as in the regular random graphs, whereas they all do well when the input is irregular, as in the random graphs with heavy components. Intuitively, this behavior is in accord with the nature of the algorithm: irregular graphs have "extra" edges that will not be in a sparse certificate and therefore be contracted. This family shows just how dramatic the difference is.

It is also interesting that excess detection appears to be more effective when the graph has extra edges, and the PR tests are not.

BIKEWHE

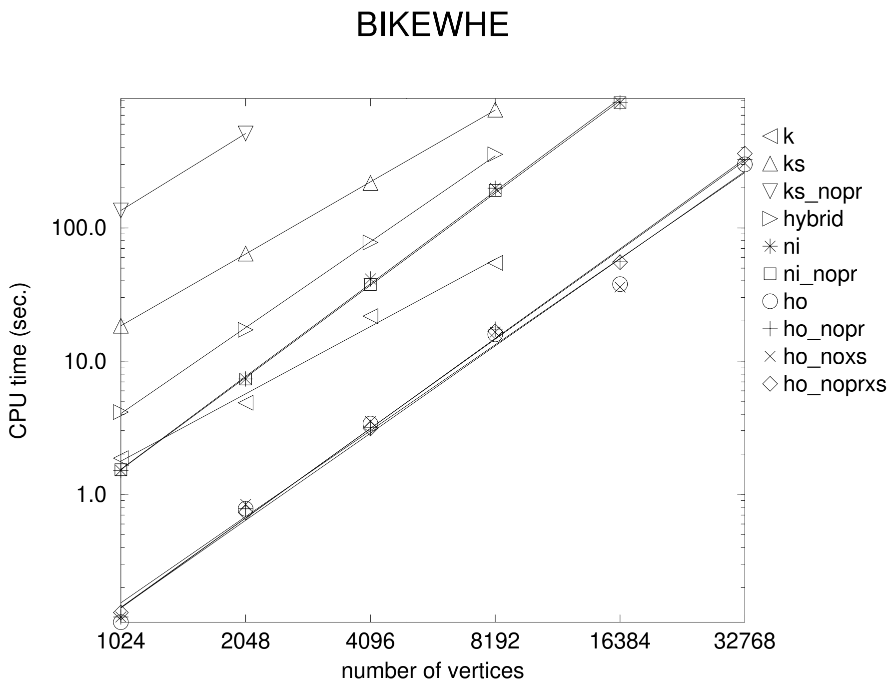

Figure 4.15: All implementations on bicycle wheel graphs.

The bicycle wheel graphs (Figure 4.15 and Table 4.7) are particularly interesting because they are the only example we have where both deterministic algorithms lose asymptotically. Unfortunately, because `k` needs $\Omega(n \log n)$ space, 128M of RAM was not sufficient memory for us to run a big enough instance to see `k` win. It appears that the crossover would take place right near the edge of the plot though.

DBLCYC

Two interleaved cycles (Figure 4.15 and Table 4.7) have interesting properties with respect to the PR tests. The graphs have $\Omega(n^2)$ near-minimum cuts, and random choices of edges to contract will not distinguish very near minimum cuts from minimum cuts, so KS should find them all. However, the PR tests can tell the difference, and this makes for a huge difference between `ks` and `ks_nopr`. `ks` runs quite well on these graphs, whereas `ks_nopr` runs so badly that we had to run it on some smaller instances in order to get any idea of how it behaved.

Surprisingly, similar behavior occurs in the implementations of NI. `ni` does a few sparse certificate computations, and then the PR tests finish off the graph. `hybrid`'s PR strategy is not nearly as

	BIKEWHE
`k`	$1.57 \times 10^{-5} \times n^{1.68}$
`ks`	$7.39 \times 10^{-5} \times n^{1.79}$
`ks_nopr`	$2.24 \times 10^{-4} \times n^{1.92}$
`hybrid`	$1.43 \times 10^{-6} \times n^{2.14}$
`ni`	$1.73 \times 10^{-7} \times n^{2.31}$
`ni_nopr`	$1.81 \times 10^{-7} \times n^{2.30}$
`ho`	$4.04 \times 10^{-8} \times n^{2.17}$
`ho_nopr`	$2.80 \times 10^{-8} \times n^{2.23}$
`ho_noxs`	$5.39 \times 10^{-8} \times n^{2.14}$
`ho_noprxs`	$2.56 \times 10^{-8} \times n^{2.24}$

Table 4.7: Asymptotic behavior of the implementations on bicycle wheel graphs.

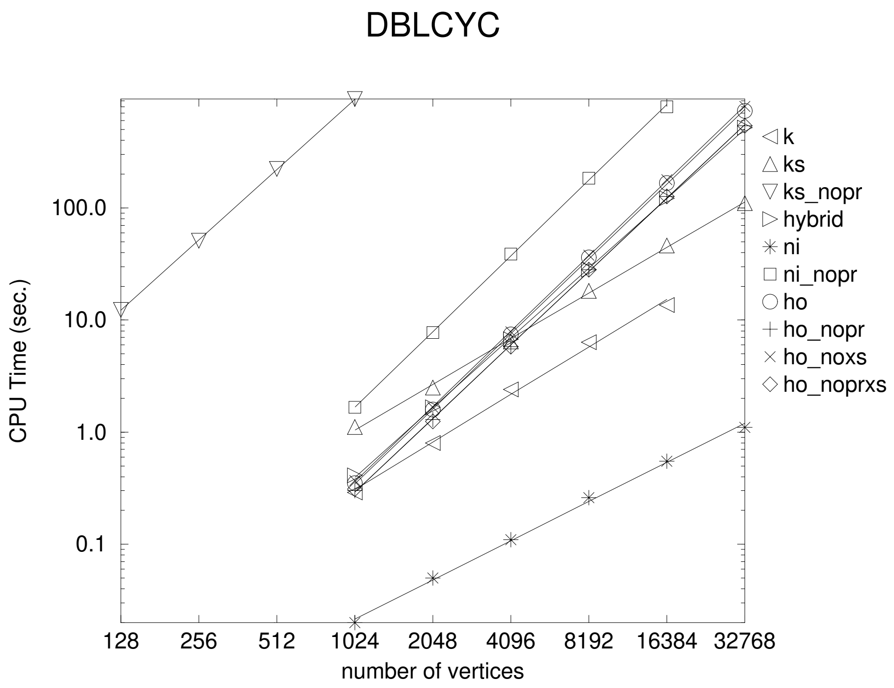

Figure 4.16: All implementations on two interleaved cycles.

	DBLCYC
k	$1.77 \times 10^{-5} \times n^{1.41}$
ks	$9.02 \times 10^{-5} \times n^{1.35}$
ks_nopr	$4.92 \times 10^{-4} \times n^{2.09}$
hybrid	$2.44 \times 10^{-7} \times n^{2.06}$
ni	$7.07 \times 10^{-6} \times n^{1.16}$
ni_nopr	$3.04 \times 10^{-7} \times n^{2.24}$
ho	$7.42 \times 10^{-8} \times n^{2.22}$
ho_nopr	$8.23 \times 10^{-8} \times n^{2.18}$
ho_noxs	$7.18 \times 10^{-8} \times n^{2.23}$
ho_noprxs	$8.36 \times 10^{-8} \times n^{2.17}$

Table 4.8: Asymptotic behavior of the implementations on two interleaved cycles.

effective, allowing roughly $n/3$ sparse certificates to be computed before finishing off the graph. This difference leads to a significant difference in asymptotic performance.

The second surprise about this family is that all of the implementations of HO perform badly. This family is the only one where `ho` does so badly. (Even on bicycle wheels, where `ho` was distinctly losing to K asymptotically, it was still winning for all of the problem sizes we ran.) Note that it is not obvious how to get `ni`'s good performance here by some kind of sparsification preprocessing. `ni` takes a few certificate computations, so repeating sparse certificate computations while they help would not work here. Further, using Matula's $2 + \epsilon$ approximation and using the resulting cut bound to compute a sparse certificate (as `k` does) reduces the problem but does not solve it.

So it appears that this family has the property that the source PR tests do not kick in for a long time, whereas PR passes seem to help very soon. It would be nice to implement PR passes in `ho` to verify this conjecture. (Note that implementing PR passes in HO is non-trivial, because a PR pass may invalidate the distance labeling.)

PR

Our only goal in running these tests used by Padberg and Rinaldi was to compare our PR strategy to theirs. We refer to their code as `pr`, and look at the number of s-t cuts computed. Since our implementations contract nodes via other heuristics, we look at all of the variants (Table 4.2.1).

It is somewhat difficult to make a meaningful comparison here because of HO's other heuristics. Looking just at the number of s-t cuts computed by `ho`, it appears that we cannot hope to do much better. However, `ho_noxs` shows that in some cases we are not getting as much out of the PR tests as `pr`. It remains unclear whether we should be concerned about this fact or not. The question is whether there are times when we would miss PR tests and excess detection would not make up the difference. It is possible that two interleaved cycles are such a case, but we have yet

n	d	c	pr	ho	ho_noxs	ho_nopr	ho_noprxs
100	2	1	3.8	1.0	1.0	1.0	55.6
200	2	1	1.8	1.0	1.0	1.0	62.8
300	2	1	1.2	1.0	1.0	1.2	62.8
400	2	1	1.0	1.6	3.8	2.2	63.4
100	10	1	3.8	1.2	1.8	2.2	37.4
200	10	1	12.6	1.8	12.4	5.0	32.4
300	10	1	23.0	2.0	11.2	5.0	48.6
400	10	1	42.8	3.4	7.6	7.6	51.2
100	50	1	1.4	1.2	2.0	5.8	26.2
200	50	1	1.6	4.0	7.4	8.0	40.4
300	50	1	2.2	5.8	9.4	9.0	50.6
400	50	1	2.4	5.0	10.6	13.8	67.6
100	100	1	1.0	1.0	1.2	7.0	26.0
200	100	1	1.0	1.0	1.4	10.4	38.4
300	100	1	1.0	1.0	1.4	11.2	55.6
400	100	1	1.0	1.0	1.2	13.6	65.8
100	2	2	6.2	1.0	1.0	3.2	68.6
200	2	2	1.6	1.0	1.0	2.2	98.8
300	2	2	1.2	1.0	1.0	1.8	115.4
400	2	2	1.0	1.0	1.4	1.8	119.2
500	2	2	1.0	1.0	2.4	2.2	126.2
1000	2	2	6.4	2.2	16.0	4.0	134.0
1500	2	2	11.8	5.8	58.8	5.8	133.0
2000	2	2	215.4	8.0	58.8	10.2	189.8
100	10	2	2.8	1.0	2.6	2.6	44.6
200	10	2	2.4	1.0	3.0	3.2	52.6
300	10	2	12.6	3.6	16.4	5.8	58.0
400	10	2	34.0	2.2	19.8	5.2	81.8
100	50	2	1.0	2.2	6.0	5.6	41.0
200	50	2	1.0	5.0	19.8	9.6	71.8
300	50	2	1.4	6.4	36.8	10.0	108.2
400	50	2	2.0	4.2	25.0	10.4	134.6
100	100	2	1.0	2.6	6.2	7.4	52.8
200	100	2	1.0	3.0	9.0	10.8	108.4
300	100	2	1.0	1.0	1.0	13.4	156.2
400	100	2	1.0	6.2	41.8	13.2	195.6

Table 4.9: Number of s-t cuts performed by HO implementations on PR graphs.

to verify that.

4.2.2 Results by Algorithm

We now shift our focus and discuss the algorithms individually. We also discuss the PR tests, as we feel that they deserve further discussion.

The Padberg-Rinaldi Heuristics

The PR heuristics are powerful, but subtle. In general, our pass strategy appears to be a good way to apply the tests. It never significantly slows down the implementation, and it often results in substantial improvements. Preprocessing with PR passes practically trivializes the TSP instances, and solves cycles, and wagon wheels, and random graphs with many heavy components. Passes also significantly improve the asymptotic behavior of `ni` and `ks` on two interleaved cycles.

We conclude that omitting PR tests from an implementation of a minimum cut algorithm would be a serious mistake.

The Hao-Orlin Algorithm

HO appears to be the best general purpose algorithm. It loses asymptotically on bicycle wheels, but it has the best performance for the size graphs we ran. The only bad case is two interleaved cycles, where it loses to all the other algorithms. Otherwise, it performs very well.

Notice that we do not have any dense families on which `ho` exhibits its worst case $\Omega(n^2\sqrt{m})$ time behavior. The worst we see is $\Omega(n^2)$ behavior on bicycle wheels and two interleaved cycles. In the maximum flow context, there are parameterized worst case instances. However, these instances have many degree two nodes, which PR2 would promptly remove, so we did not try them. It would be nice to either find a worst case family or prove that the PR tests improve HO's time bounds.

In general, `ho` is so fast on its own that the heuristics do not help a great deal. On the TSP instances, PR preprocessing gains an order of magnitude, but that is the most extreme example. The heuristics also "compete" with each other for contractions. Comparing `ho`, `ho_noxs`, and `ho_nopr`, we see that they all do similar numbers of heuristic contractions. Both the PR tests and excess detection can be responsible for many heuristic contractions, but when we put them together a contraction done by one typically means one less done by the other. Further, without either heuristic, we get many one node layers, so the "flow computation" we are saving is often trivial.

It is valuable to do PR preprocessing, but the value of internal PR tests is unclear. One direction we see that should be explored is the idea of doing PR passes periodically. Such a strategy may improve `ho`'s bad performance on two interleaved cycles.

The behavior of excess detection is interesting. Both NOI6 and IRREG suggest that excess detection is good at identifying extra edges, in a way that the PR tests are not. It would be nice to establish a stronger statement on the relationship between excess detection and NI.

The Nagamochi-Ibaraki Algorithm

Results on implementations of NI are mixed. When `ni` works well, it works very well, but sometimes it does show its worst case behavior. In the long run, the sparse certificate computation may be more valuable than the whole algorithm. That is, the good case for `ni` is when the sparse certificate exhibits many extra edges, and this computation can be done as a preprocessing step for any other algorithm. Such a strategy would take advantage of `ni`'s performance in the good cases and resort to another algorithm for the bad ones. The inclusion of this strategy is partially responsible for the reasonable performance of `k`.

`ni` compares favorably with `hybrid`. Our implementation never loses by much, and sometimes our improved PR strategy gives us much better performance. In particular, we win by about three orders of magnitude on TSP instances, and we have better (by a factor of n) asymptotic performance on two interleaved cycles.

The Karger-Stein Algorithm

`ks` seems to be inferior to the other implementations, but it is not clear that this statement can be generalized to KS. We have no problem families on which `ks` wins, but on bicycle wheels it only loses to `k`, which is cheating on its probabilities.

For many families, it is not clear that the internal PR tests help a great deal, but they help asymptotically on two interleaved cycles, so we conclude that they are valuable. Note PR tests in general help less in `ks` than in other codes, because we may have to undo them. That is, when we apply PR tests after random contractions, they may pass only because we have already contracted a minimum cut edge, so when we undo the random contractions (backing out of the recursion), we must undo the PR contractions as well. As a result, `ks` often does more PR contractions than there are nodes in the graph.

The frustrating thing about implementing KS is its Monte Carlo nature—correctness depends on the analysis, and tight analysis is difficult. Our evidence on tightness is mixed. For several random graphs with heavy components, it occasionally happens that the discovery time is a substantial fraction (more than 20%) of the post-preprocessing time, which says that if we ran many fewer recursion trees we would get the wrong answer. However, we only picked the constants to guarantee a 95% success probability, so we should expect to get the wrong answer occasionally. Even on the problems where we see high discovery times, most of the discovery times are still small. So there is no hope of improving `ks` by many orders of magnitude, but improvement by as much as one order of magnitude looks plausible.

The other evidence that `ks` can improve is the fact that the trees are almost always far shallower than the theory predicts, which suggests that the probability of finding the minimum cut is higher than the theory estimates. However, even though the theory analysis gives us an estimate of success probability based on depth, we cannot estimate the probability after looking at the depth of the tree, because conditioning on a shallow depth, it becomes more likely that we contracted a minimum cut edge. This state is particularly frustrating because the PR tests, which do not make mistakes, are liable to be responsible for the shallow depth, but we do not know how to tell the difference on-line.

It seems that there should be a way to estimate the probability of success on-line, based on what the algorithm has done, and get tighter estimates than the off-line analysis would give, but we have not been able to find it.

A heuristic that might help `ks`, but that we did not get a chance to try, is to do a sparse certificate computation after the random contractions. If the random contractions tend to create graphs with extra edges, this strategy may help to reduce the depth of the recursion.

Another possible heuristic that should be explored is deliberately overestimating the minimum cut so that the success probability of each recursion tree increases. Preliminary experiments with this idea suggested that it does speed up the implementation, but careful analysis of the running time is needed to make sure that such an implementation will not have termination problems.

Karger's Algorithm

Results on K remain inconclusive. The performance of `k` is not amazing, but it is not terrible either. Of course, since we cheated on the probabilities, it is not clear that timings on `k` mean anything, but since `k` never got the wrong answer, we believe that they do. It would be nice to either find a problem family that demonstrates the tightness of the theory results or improve the bound.

Note that two interleaved cycles were originally designed as an attempt to stress the sampling probabilities. They have one minimum cut and $\Omega(n^2)$ near minimum cuts, so they should be in danger of generating a sampled graph in which a tree packing might have no trees that the minimum cut 2-respects. However, in 70,000 runs on a 1000 node double cycle, we always found a tree that the minimum cut 2-respected. The reason for success, though, seems to be that when the sampled graph looks roughly like a cycle, there is little question that at least one tree will be path going around it. We are not sure if this observation can be exploited at all to tighten the analysis.

Tree packing is often the bottleneck, so it would also be valuable to work more on improving Gabow's algorithm. With a theoretically justifiable sampling probability, the problem we have is that Gabow's algorithm needs to explicitly represent the trees, and there are too many. Perhaps some kind of scaling approach could work around this problem. Karger's divide and conquer approach [34], which improves the asymptotic running time to $O(\sqrt{\lambda} m \log n)$, might also improve performance.

The major implementation question that remains to be answered is what happens if we use the fractional packing algorithm of Plotkin, Shmoys, and Tardos to compute the tree packings instead of Gabow's algorithm. There are two reasons why PST might be better. First, it does not need an explicit representation of the trees, so it will demand less memory, and may allow us to use theoretically justifiable sampling probabilities. Second, it is possible that it will often find more than the minimum number of trees, in which case we would be able to check only for 1-respecting cuts.

Chapter 5

Conclusion

Our study produced several efficient implementations of minimum cut algorithms, improving the previous state of the art. We introduced new strategies for improving performance, and we give several problem families on which future implementations can be tested.

Our tests show that no single algorithm dominates the others. `ho` and `ni` typically dominate `ks` and `k`, but on bicycle wheels, asymptotically the reverse is true. `ho` and `ni` are hard to compare directly; on regular random graphs `ho` does well and `ni` does poorly, and on two interleaved cycles `ni` does well and `ho` does poorly. For general purposes we would recommend `ho`, as it has such small constant factors that even when it performs "badly", it does pretty well. After `ho` we would recommend `ni`.

Our results confirm the importance of the Padberg-Rinaldi heuristics. In some cases they improve the practical performance by several orders of magnitude, and in other cases they clearly improve the asymptotic performance of an implementation. We conclude that omitting PR tests from an implementation of a minimum cut algorithm would be a serious mistake. It is said that implementations using graph contraction are usually difficult to code (see *e.g.* [28]) and may be inefficient, but the gains of the Padberg-Rinaldi heuristics easily make contraction worth implementing.

There are several possible directions for future work. On the implementation side, there are a few possibilities that should be explored. First, further experiments on using PR tests in `ho` should be done. In particular, we would like to know if PR passes will cure the bad behavior on two interleaved cycles. Second, some experiments on adding sparse certificate computations to the other algorithms should be done. Sparse certificates might help `ho` a little bit, and it would be interesting to see if they help inside the recursion of `ks`. Finally, PST tree packing should be tried in `k`. This change might make it feasible for `k` to use theoretically justifiable sampling probabilities, as well as possibly improving performance. It would also be nice to try to improve Gabow's algorithm, possibly with some further heuristics.

There are also some open questions on the theory side. First, either a graph that causes worst-case behavior in HO should be found, or it should be proved that the addition of heuristics actually improves the time bounds. Second, both KS and K would benefit from more theory work. Since they are Monte Carlo in nature, we can only guarantee "correctness" by using the theoretical analysis, and we do not believe that the constant factors of the analysis are tight in either case. The idea of overestimating the minimum cut in KS should be explored, and it would be nice to get an analysis of KS that allowed for a more on-line estimation of success probabilities.

Appendix A

Data Tables

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	532	787	0.01	0.00	0.00	0.00	10072	0.00	0.00	487	14	9	16.67	7	13	-	-	-	-
ho_nopr	532	787	0.03	0.00	0.02	0.00	62950	0.00	-	-	-	65	54.43	15	451	-	-	-	-
ho_noprxs	532	787	0.03	0.00	0.02	0.00	73118	0.00	-	-	-	480	15.58	51	-	-	-	-	-
ho_noxs	532	787	0.01	0.00	0.00	0.00	10250	0.00	0.00	487	13	21	10.29	9	-	-	-	-	-
hybrid	532	787	0.51	0.00	-	-	-	-	-	-	136	-	-	-	-	-	-	-	-
k	532	787	0.01	0.00	0.00	0.00	10043	0.00	0.00	512	17	-	-	-	-	-	-	-	-
ks	532	787	0.03	0.31	0.00	0.00	16077	0.03	0.00	512	405	-	-	-	-	-	331	-	-
ks_nopr	532	787	0.07	0.11	0.00	0.00	18375	0.03	0.00	512	0	-	-	-	-	-	1676	-	-
ni	532	787	0.01	0.00	0.00	0.00	2350	0.00	0.00	487	37	-	-	-	-	4	-	-	-
ni_nopr	532	787	0.46	0.00	0.44	0.00	698119	0.00	-	-	-	-	-	-	-	491	-	-	-
ho	1748	2336	0.02	0.00	0.01	0.00	39701	0.00	0.00	1611	25	35	31.86	29	47	-	-	-	-
ho_nopr	1748	2336	0.12	0.00	0.12	0.00	333314	0.00	-	-	-	213	126.02	55	1479	-	-	-	-
ho_noprxs	1748	2336	0.11	0.00	0.11	0.00	298036	0.00	-	-	-	1623	39.59	124	-	-	-	-	-
ho_noxs	1748	2336	0.01	0.00	0.01	0.00	40028	0.00	0.01	1611	42	62	19.15	31	-	-	-	-	-
hybrid	1748	2336	4.67	0.00	-	-	-	-	-	-	535	-	-	-	-	-	-	-	-
k	1748	2336	0.07	0.30	0.01	0.00	90134	0.32	0.01	1671	23	-	-	-	-	-	-	0.03	0.02
ks	1748	2336	0.18	0.05	0.01	0.50	87416	0.02	0.01	1671	2790	-	-	-	-	-	1107	-	-
ks_nopr	1748	2336	0.43	0.12	0.01	0.00	106655	0.04	0.01	1671	0	-	-	-	-	-	9901	-	-
ni	1748	2336	0.02	0.00	0.00	0.00	17124	0.00	0.01	1611	98	-	-	-	-	28	-	-	-
ni_nopr	1748	2336	4.75	0.00	4.61	0.00	7123685	0.00	-	-	-	-	-	-	-	1576	-	-	-
ho	15112	19057	0.27	0.00	0.14	0.00	443297	0.00	0.12	13912	147	290	42.64	269	493	-	-	-	-
ho_nopr	15112	19057	2.89	0.00	2.74	0.00	5880926	0.00	-	-	-	1910	107.13	335	12866	-	-	-	-
ho_noprxs	15112	19057	2.97	0.00	2.79	0.00	6117231	0.00	-	-	-	14081	26.59	1030	-	-	-	-	-
ho_noxs	15112	19057	0.26	0.00	0.13	0.00	448498	0.00	0.11	13912	150	692	19.26	356	-	-	-	-	-
hybrid	15112	19057	301.19	0.00	-	-	-	-	-	-	1948	-	-	-	-	-	-	-	-
k	15112	19057	2.50	0.14	0.14	0.03	1811697	0.19	0.16	14473	151	-	-	-	-	-	-	1.95	0.33
ks	15112	19057	2.01	0.01	0.11	0.00	924550	0.00	0.13	14473	28303	-	-	-	-	-	2537	-	-
ks_nopr	15112	19057	4.19	0.02	0.11	0.04	1069057	0.02	0.13	14473	0	-	-	-	-	-	60392	-	-
ni	15112	19057	0.29	0.00	0.12	0.00	189001	0.00	0.11	13912	775	-	-	-	-	64	-	-	-
ni_nopr	15112	19057	311.42	0.00	205.75	0.00	226365793	0.00	-	-	-	-	-	-	-	7834	-	-	-
ho	33810	38600	0.27	0.00	0.27	0.00	271169	0.00	0.18	33808	0	1	2.00	0	0	-	-	-	-
ho_nopr	33810	38600	4.60	0.00	4.49	0.00	8708022	0.00	-	-	-	151	2121.44	12	33646	-	-	-	-
ho_noprxs	33810	38600	7.34	0.00	5.25	0.00	14351590	0.00	-	-	-	32201	165.57	1608	-	-	-	-	-
ho_noxs	33810	38600	0.27	0.00	0.27	0.00	271169	0.00	0.18	33808	0	1	2.00	0	-	-	-	-	-
hybrid	33810	38600	1.77	0.00	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	33810	38600	0.31	0.02	0.31	0.02	271168	0.00	0.23	33808	0	-	-	-	-	-	-	-	-
ks	33810	38600	0.23	0.00	0.23	0.02	193968	0.00	0.17	33808	-	-	-	-	-	-	0	-	-
ks_nopr	33810	38600	0.23	0.00	0.23	0.00	193968	0.00	0.17	33808	-	-	-	-	-	-	0	-	-
ni	33810	38600	0.22	0.00	0.22	0.00	77200	0.00	0.16	33808	0	-	-	-	-	0	-	-	-
ni_nopr	33810	38600	0.54	0.00	0.54	0.00	395035	0.00	-	-	-	-	-	-	-	6	-	-	-
ho	33810	39367	0.30	0.00	0.30	0.00	303275	0.00	0.21	33808	0	1	2.00	0	0	-	-	-	-
ho_nopr	33810	39367	5.08	0.00	5.02	0.00	8077063	0.00	-	-	-	283	4226.33	51	33475	-	-	-	-
ho_noprxs	33810	39367	11.71	0.00	8.44	0.00	19261846	0.00	-	-	-	31796	707.20	2013	-	-	-	-	-
ho_noxs	33810	39367	0.30	0.00	0.30	0.00	303275	0.00	0.21	33808	0	1	2.00	0	-	-	-	-	-
hybrid	33810	39367	6.27	0.00	-	-	-	-	-	-	320	-	-	-	-	-	-	-	-
k	33810	39367	0.33	0.01	0.33	0.01	303274	0.00	0.26	33808	0	-	-	-	-	-	-	-	-
ks	33810	39367	0.25	0.02	0.25	0.00	224540	0.00	0.19	33808	-	-	-	-	-	-	0	-	-
ks_nopr	33810	39367	0.25	0.02	0.24	0.02	224540	0.00	0.19	33808	-	-	-	-	-	-	0	-	-
ni	33810	39367	0.26	0.00	0.25	0.00	78734	0.00	0.19	33808	0	-	-	-	-	0	-	-	-
ni_nopr	33810	39367	2.96	0.00	2.96	0.00	2163704	0.00	-	-	-	-	-	-	-	17	-	-	-
ho	33810	39456	0.30	0.00	0.28	0.00	295708	0.00	0.20	33808	0	1	2.00	0	0	-	-	-	-
ho_nopr	33810	39456	3.70	0.00	3.13	0.00	6756961	0.00	-	-	-	368	1257.12	53	33388	-	-	-	-
ho_noprxs	33810	39456	8.81	0.00	4.78	0.00	16069243	0.00	-	-	-	31845	305.51	1964	-	-	-	-	-
ho_noxs	33810	39456	0.29	0.00	0.27	0.00	295708	0.00	0.20	33808	0	1	2.00	0	-	-	-	-	-
hybrid	33810	39456	2.24	0.00	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	33810	39456	0.33	0.00	0.31	0.02	295707	0.00	0.25	33808	0	-	-	-	-	-	-	-	-
ks	33810	39456	0.24	0.00	0.22	0.02	216795	0.00	0.18	33808	-	-	-	-	-	-	0	-	-
ks_nopr	33810	39456	0.24	0.03	0.22	0.02	216795	0.00	0.18	33808	-	-	-	-	-	-	0	-	-
ni	33810	39456	0.25	0.00	0.23	0.00	78912	0.00	0.19	33808	0	-	-	-	-	0	-	-	-
ni_nopr	33810	39456	1.20	0.00	1.19	0.00	872145	0.00	-	-	-	-	-	-	-	12	-	-	-

Table A.1: TSP data

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	85900	102596	0.87	0.00	0.79	0.00	810397	0.00	0.64	85898	0	1	2.00	0	0	-	-	-	-
ho_nopr	85900	102596	4.20	0.00	3.53	0.00	7014415	0.00	-	-	-	202	3915.79	21	85676	-	-	-	-
ho_noprxs	85900	102596	34.54	0.00	6.06	0.00	63960815	0.00	-	-	-	79725	248.18	6174	-	-	-	-	-
ho_noxs	85900	102596	0.85	0.00	0.78	0.00	810397	0.00	0.62	85898	0	1	2.00	0	-	-	-	-	-
hybrid	85900	102596	8.97	0.00	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-
k	85900	102596	0.99	0.00	0.90	0.00	810396	0.00	0.79	85898	0	-	-	-	-	-	-	-	-
ks	85900	102596	0.73	0.00	0.66	0.00	605204	0.00	0.57	85898	-	-	-	-	-	-	0	-	-
ks_nopr	85900	102596	0.72	0.01	0.65	0.01	605204	0.00	0.57	85898	-	-	-	-	-	-	0	-	-
ni	85900	102596	0.73	0.00	0.66	0.00	205192	0.00	0.57	85898	0	-	-	-	-	0	-	-	-
ni_nopr	85900	102596	3.83	0.00	3.83	0.00	2704804	0.00	-	-	-	-	-	-	-	9	-	-	-
ho	85900	102934	0.90	0.00	0.90	0.00	855359	0.00	0.67	85898	0	1	2.00	0	0	-	-	-	-
ho_nopr	85900	102934	11.12	0.00	11.04	0.00	18125856	0.00	-	-	-	477	6293.96	134	85288	-	-	-	-
ho_noprxs	85900	102934	33.08	0.00	32.12	0.00	55749839	0.00	-	-	-	79900	241.92	5999	-	-	-	-	-
ho_noxs	85900	102934	0.88	0.00	0.87	0.00	855359	0.00	0.66	85898	0	1	2.00	0	-	-	-	-	-
hybrid	85900	102934	11.03	0.00	-	-	-	-	-	-	22	-	-	-	-	-	-	-	-
k	85900	102934	1.02	0.00	1.02	0.00	855358	0.00	0.82	85898	0	-	-	-	-	-	-	-	-
ks	85900	102934	0.76	0.01	0.76	0.01	649490	0.00	0.60	85898	-	-	-	-	-	-	0	-	-
ks_nopr	85900	102934	0.75	0.01	0.74	0.01	649490	0.00	0.60	85898	-	-	-	-	-	-	0	-	-
ni	85900	102934	0.74	0.00	0.74	0.00	205868	0.00	0.58	85898	0	-	-	-	-	0	-	-	-
ni_nopr	85900	102934	4.94	0.00	4.94	0.00	3499861	0.00	-	-	-	-	-	-	-	13	-	-	-
ho	85900	102988	0.91	0.00	0.33	0.00	869143	0.00	0.68	85830	3	4	47.50	0	61	-	-	-	-
ho_nopr	85900	102988	20.28	0.00	11.72	0.00	34575949	0.00	-	-	-	674	5491.61	160	85065	-	-	-	-
ho_noprxs	85900	102988	32.71	0.00	12.06	0.00	59777886	0.00	-	-	-	79931	209.54	5968	-	-	-	-	-
ho_noxs	85900	102988	0.89	0.00	0.32	0.00	870001	0.00	0.66	85830	46	13	22.85	9	-	-	-	-	-
hybrid	85900	102988	10.96	0.00	-	-	-	-	-	-	11	-	-	-	-	-	-	-	-
k	85900	102988	1.03	0.00	0.31	0.01	870500	0.00	0.84	85839	22	-	-	-	-	-	-	-	-
ks	85900	102988	0.86	0.01	0.25	0.00	690944	0.00	0.61	85839	1551	-	-	-	-	-	599	-	-
ks_nopr	85900	102988	0.95	0.02	0.25	0.02	698830	0.00	0.61	85839	0	-	-	-	-	-	4470	-	-
ni	85900	102988	0.80	0.00	0.25	0.00	206642	0.00	0.64	85830	6	-	-	-	-	2	-	-	-
ni_nopr	85900	102988	4.96	0.00	3.99	0.00	3502182	0.00	-	-	-	-	-	-	-	16	-	-	-
ho	13509	15631	0.14	0.00	0.14	0.00	196893	0.00	0.09	12976	70	80	43.86	69	313	-	-	-	-
ho_nopr	13509	15631	2.32	0.00	1.35	0.00	4613412	0.00	-	-	-	803	261.11	127	12578	-	-	-	-
ho_noprxs	13509	15631	2.47	0.00	1.39	0.00	4897923	0.00	-	-	-	12733	41.52	775	-	-	-	-	-
ho_noxs	13509	15631	0.17	0.00	0.17	0.00	248615	0.00	0.08	12976	180	240	47.70	111	-	-	-	-	-
hybrid	13509	15631	124.12	0.00	-	-	-	-	-	-	1222	-	-	-	-	-	-	-	-
k	13509	15631	0.65	0.42	0.13	0.00	537593	0.39	0.11	13185	73	-	-	-	-	-	-	0.39	0.11
ks	13509	15631	0.58	0.02	0.10	0.05	301960	0.02	0.08	13185	8471	-	-	-	-	-	798	-	-
ks_nopr	13509	15631	0.82	0.03	0.10	0.04	286172	0.02	0.08	13185	0	-	-	-	-	-	7023	-	-
ni	13509	15631	0.13	0.00	0.12	0.00	55118	0.00	0.07	12976	270	-	-	-	-	18	-	-	-
ni_nopr	13509	15631	90.19	0.00	9.84	0.00	72585922	0.00	-	-	-	-	-	-	-	4142	-	-	-
ho	13509	17048	0.23	0.00	0.22	0.00	375035	0.00	0.10	12492	163	289	43.43	200	364	-	-	-	-
ho_nopr	13509	17048	2.86	0.00	2.45	0.00	5877064	0.00	-	-	-	1580	113.33	292	11636	-	-	-	-
ho_noprxs	13509	17048	2.97	0.00	2.48	0.00	6085267	0.00	-	-	-	12591	34.71	917	-	-	-	-	-
ho_noxs	13509	17048	0.24	0.00	0.23	0.00	419222	0.00	0.10	12492	155	562	26.27	298	-	-	-	-	-
hybrid	13509	17048	314.77	0.00	-	-	-	-	-	-	2591	-	-	-	-	-	-	-	-
k	13509	17048	0.85	0.15	0.17	0.03	667280	0.14	0.14	12985	237	-	-	-	-	-	-	0.46	0.20
ks	13509	17048	1.72	0.01	0.20	0.13	784104	0.01	0.11	12985	23607	-	-	-	-	-	2259	-	-
ks_nopr	13509	17048	3.53	0.02	0.75	0.79	916754	0.01	0.11	12985	0	-	-	-	-	-	51033	-	-
ni	13509	17048	0.25	0.00	0.17	0.00	171263	0.00	0.09	12492	685	-	-	-	-	62	-	-	-
ni_nopr	13509	17048	333.05	0.00	28.94	0.00	255366401	0.00	-	-	-	-	-	-	-	9033	-	-	-
ho	1291	1942	0.02	0.00	0.00	0.00	64813	0.00	0.00	976	45	83	25.65	59	127	-	-	-	-
ho_nopr	1291	1942	0.11	0.00	0.11	0.00	297504	0.00	-	-	-	215	63.07	90	985	-	-	-	-
ho_noprxs	1291	1942	0.11	0.00	0.11	0.00	308678	0.00	-	-	-	1143	19.37	147	-	-	-	-	-
ho_noxs	1291	1942	0.02	0.00	0.00	0.00	66384	0.00	0.00	976	34	189	13.91	90	-	-	-	-	-
hybrid	1291	1942	2.94	0.00	-	-	-	-	-	-	326	-	-	-	-	-	-	-	-
k	1291	1942	0.16	0.24	0.00	2.00	222684	0.28	0.01	1170	36	-	-	-	-	-	-	0.10	0.04
ks	1291	1942	0.36	0.02	0.01	0.82	157836	0.01	0.01	1170	5134	-	-	-	-	-	1665	-	-
ks_nopr	1291	1942	0.86	0.05	0.00	1.22	197768	0.02	0.01	1170	0	-	-	-	-	-	17772	-	-
ni	1291	1942	0.03	0.00	0.01	0.00	31238	0.00	0.01	976	260	-	-	-	-	28	-	-	-
ni_nopr	1291	1942	3.09	0.00	3.04	0.00	4655872	0.00	-	-	-	-	-	-	-	1213	-	-	-

Table A.2: TSP data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	13509	17079	0.23	0.00	0.16	0.00	376562	0.00	0.10	12456	176	254	43.64	238	384	-	-	-	-
ho_nopr	13509	17079	3.04	0.00	1.05	0.00	6324104	0.00	-	-	-	1662	118.78	284	11562	-	-	-	-
ho_noprxs	13509	17079	3.04	0.00	1.06	0.00	6222631	0.00	-	-	-	12615	35.91	893	-	-	-	-	-
ho_noxs	13509	17079	0.22	0.00	0.15	0.00	377492	0.00	0.10	12456	187	544	23.46	320	-	-	-	-	-
hybrid	13509	17079	263.55	0.00	-	-	-	-	-	-	2099	-	-	-	-	-	-	-	-
k	13509	17079	1.34	0.15	1.12	0.18	1045496	0.19	0.13	13003	149	-	-	-	-	-	-	0.93	0.23
ks	13509	17079	1.76	0.01	0.33	0.49	807922	0.01	0.11	13003	24340	-	-	-	-	-	2490	-	-
ks_nopr	13509	17079	3.99	0.01	0.47	0.66	984983	0.01	0.11	13003	0	-	-	-	-	-	64745	-	-
ni	13509	17079	0.25	0.00	0.21	0.00	174235	0.00	0.08	12456	700	-	-	-	-	66	-	-	-
ni_nopr	13509	17079	315.88	0.00	275.94	0.00	242984346	0.00	-	-	-	-	-	-	-	8918	-	-	-
ho	13509	17111	0.20	0.00	0.02	0.00	305102	0.00	0.11	12707	123	216	35.69	196	266	-	-	-	-
ho_nopr	13509	17111	2.78	0.00	0.01	0.00	5686705	0.00	-	-	-	1724	135.93	305	11479	-	-	-	-
ho_noprxs	13509	17111	2.88	0.00	0.01	0.00	5953999	0.00	-	-	-	12607	29.47	901	-	-	-	-	-
ho_noxs	13509	17111	0.20	0.00	0.02	0.00	323479	0.00	0.10	12707	144	409	18.11	247	-	-	-	-	-
hybrid	13509	17111	296.88	0.00	-	-	-	-	-	-	2140	-	-	-	-	-	-	-	-
k	13509	17111	0.88	0.30	0.01	0.00	719920	0.25	0.14	13021	188	-	-	-	-	-	-	0.50	0.20
ks	13509	17111	1.66	0.01	0.01	0.82	763462	0.00	0.11	13021	22371	-	-	-	-	-	2492	-	-
ks_nopr	13509	17111	3.46	0.02	0.01	0.50	895027	0.01	0.11	13021	0	-	-	-	-	-	53540	-	-
ni	13509	17111	0.25	0.00	0.00	0.00	161549	0.00	0.10	12707	506	-	-	-	-	64	-	-	-
ni_nopr	13509	17111	303.10	0.00	0.00	0.00	230590104	0.00	-	-	-	-	-	-	-	8560	-	-	-
ho	13509	17130	0.19	0.00	0.18	0.00	275500	0.00	0.11	12725	97	79	64.90	95	512	-	-	-	-
ho_nopr	13509	17130	2.61	0.00	2.60	0.00	5396663	0.00	-	-	-	777	195.55	172	12559	-	-	-	-
ho_noprxs	13509	17130	2.77	0.00	1.30	0.00	5627773	0.00	-	-	-	12558	32.31	950	-	-	-	-	-
ho_noxs	13509	17130	0.20	0.00	0.17	0.00	317520	0.00	0.11	12725	271	328	26.79	183	-	-	-	-	-
hybrid	13509	17130	33.68	0.00	-	-	-	-	-	-	662	-	-	-	-	-	-	-	-
k	13509	17130	0.57	0.17	0.16	0.00	475887	0.13	0.14	13065	179	-	-	-	-	-	-	0.28	0.11
ks	13509	17130	1.43	0.01	0.13	0.04	671141	0.01	0.11	13065	19603	-	-	-	-	-	2110	-	-
ks_nopr	13509	17130	2.92	0.03	0.13	0.00	772748	0.01	0.11	13065	0	-	-	-	-	-	43979	-	-
ni	13509	17130	0.16	0.00	0.14	0.00	71857	0.00	0.10	12725	398	-	-	-	-	14	-	-	-
ni_nopr	13509	17130	54.42	0.00	46.69	0.00	39857283	0.00	-	-	-	-	-	-	-	1569	-	-	-
ho	13509	17156	0.22	0.00	0.19	0.00	358283	0.00	0.11	12649	162	206	39.65	203	288	-	-	-	-
ho_nopr	13509	17156	2.50	0.00	2.46	0.00	5095245	0.00	-	-	-	1735	141.48	310	11463	-	-	-	-
ho_noprxs	13509	17156	2.62	0.00	2.57	0.00	5331468	0.00	-	-	-	12605	67.04	903	-	-	-	-	-
ho_noxs	13509	17156	0.22	0.00	0.20	0.00	360071	0.00	0.11	12649	161	429	25.25	268	-	-	-	-	-
hybrid	13509	17156	297.61	0.00	-	-	-	-	-	-	2410	-	-	-	-	-	-	-	-
k	13509	17156	1.44	0.51	1.19	0.60	1150264	0.52	0.14	12948	139	-	-	-	-	-	-	0.99	0.26
ks	13509	17156	1.95	0.01	0.18	0.15	882280	0.01	0.11	12948	26923	-	-	-	-	-	2751	-	-
ks_nopr	13509	17156	4.30	0.01	0.62	0.37	1061215	0.01	0.11	12948	0	-	-	-	-	-	65466	-	-
ni	13509	17156	0.26	0.00	0.21	0.00	175560	0.00	0.10	12649	529	-	-	-	-	78	-	-	-
ni_nopr	13509	17156	286.75	0.00	247.56	0.00	222044958	0.00	-	-	-	-	-	-	-	8654	-	-	-
ho	13509	17156	0.21	0.00	0.21	0.00	331350	0.00	0.11	12684	150	189	46.72	201	283	-	-	-	-
ho_nopr	13509	17156	2.35	0.00	2.34	0.00	4892725	0.00	-	-	-	1728	103.06	303	11477	-	-	-	-
ho_noprxs	13509	17156	2.57	0.00	2.55	0.00	5262358	0.00	-	-	-	12624	34.30	884	-	-	-	-	-
ho_noxs	13509	17156	0.21	0.00	0.19	0.00	340113	0.00	0.10	12684	176	400	38.25	247	-	-	-	-	-
hybrid	13509	17156	245.46	0.00	-	-	-	-	-	-	2151	-	-	-	-	-	-	-	-
k	13509	17156	1.47	0.35	0.18	0.03	1134943	0.38	0.13	12925	227	-	-	-	-	-	-	1.04	0.24
ks	13509	17156	1.98	0.01	0.20	0.39	890033	0.01	0.11	12925	27337	-	-	-	-	-	2718	-	-
ks_nopr	13509	17156	4.22	0.01	0.39	0.40	1055280	0.01	0.11	12925	0	-	-	-	-	-	61046	-	-
ni	13509	17156	0.24	0.00	0.16	0.00	148383	0.00	0.10	12684	522	-	-	-	-	52	-	-	-
ni_nopr	13509	17156	298.84	0.00	162.38	0.00	226366501	0.00	-	-	-	-	-	-	-	7951	-	-	-
ho	13509	17183	0.21	0.00	0.12	0.00	341914	0.00	0.10	12419	110	189	43.08	157	633	-	-	-	-
ho_nopr	13509	17183	2.66	0.00	1.13	0.00	5494783	0.00	-	-	-	995	192.56	211	12302	-	-	-	-
ho_noprxs	13509	17183	2.88	0.00	1.43	0.00	5862169	0.00	-	-	-	12565	37.00	943	-	-	-	-	-
ho_noxs	13509	17183	0.27	0.00	0.12	0.00	490301	0.00	0.10	12419	224	545	31.59	319	-	-	-	-	-
hybrid	13509	17183	267.18	0.00	-	-	-	-	-	-	2122	-	-	-	-	-	-	-	-
k	13509	17183	1.98	0.30	0.13	0.04	1426615	0.34	0.14	12966	105	-	-	-	-	-	-	1.54	0.25
ks	13509	17183	1.84	0.01	0.10	0.05	831087	0.01	0.11	12966	25315	-	-	-	-	-	2479	-	-
ks_nopr	13509	17183	3.96	0.01	0.10	0.04	1013403	0.01	0.11	12966	0	-	-	-	-	-	60628	-	-
ni	13509	17183	0.21	0.00	0.09	0.00	129969	0.00	0.09	12419	678	-	-	-	-	36	-	-	-
ni_nopr	13509	17183	227.98	0.00	4.96	0.00	176354954	0.00	-	-	-	-	-	-	-	6620	-	-	-

Table A.3: TSP data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	13509	17193	0.23	0.00	0.23	0.00	394417	0.00	0.10	12427	185	286	39.33	250	360	-	-	-	-
ho_nopr	13509	17193	2.74	0.00	2.69	0.00	5762394	0.00	-	-	-	1703	109.51	321	11484	-	-	-	-
ho_noprxs	13509	17193	2.88	0.00	2.82	0.00	6027148	0.00	-	-	-	12595	33.78	913	-	-	-	-	-
ho_noxs	13509	17193	0.25	0.00	0.24	0.00	440299	0.00	0.10	12427	171	585	24.34	324	-	-	-	-	-
hybrid	13509	17193	298.50	0.00	-	-	-	-	-	-	2319	-	-	-	-	-	-	-	-
k	13509	17193	1.24	0.40	0.18	0.03	992759	0.37	0.14	12918	208	-	-	-	-	-	-	0.83	0.22
ks	13509	17193	2.13	0.01	0.24	0.18	950330	0.01	0.12	12918	28941	-	-	-	-	-	2763	-	-
ks_nopr	13509	17193	4.60	0.01	0.68	0.62	1139164	0.01	0.12	12918	0	-	-	-	-	-	69081	-	-
ni	13509	17193	0.27	0.00	0.25	0.00	183706	0.00	0.09	12427	707	-	-	-	-	66	-	-	-
ni_nopr	13509	17193	295.31	0.00	180.30	0.00	225010925	0.00	-	-	-	-	-	-	-	8667	-	-	-
ho	13509	17210	0.21	0.00	0.13	0.00	327575	0.00	0.11	12645	133	187	33.26	200	343	-	-	-	-
ho_nopr	13509	17210	2.50	0.00	1.60	0.00	5266937	0.00	-	-	-	1479	124.68	285	11744	-	-	-	-
ho_noprxs	13509	17210	2.64	0.00	1.65	0.00	5484353	0.00	-	-	-	12577	35.81	931	-	-	-	-	-
ho_noxs	13509	17210	0.21	0.00	0.13	0.00	337505	0.00	0.11	12645	179	434	21.55	249	-	-	-	-	-
hybrid	13509	17210	268.99	0.00	-	-	-	-	-	-	2300	-	-	-	-	-	-	-	-
k	13509	17210	1.36	0.23	0.14	0.00	1040112	0.22	0.14	12978	175	-	-	-	-	-	-	0.94	0.23
ks	13509	17210	1.86	0.01	0.11	0.00	848771	0.01	0.11	12978	25337	-	-	-	-	-	2575	-	-
ks_nopr	13509	17210	4.06	0.01	0.11	0.05	1018408	0.01	0.11	12978	0	-	-	-	-	-	61326	-	-
ni	13509	17210	0.27	0.00	0.11	0.00	188033	0.00	0.10	12645	534	-	-	-	-	76	-	-	-
ni_nopr	13509	17210	289.69	0.00	26.14	0.00	219639355	0.00	-	-	-	-	-	-	-	8199	-	-	-
ho	13509	17303	0.26	0.00	0.08	0.00	456907	0.00	0.10	12402	221	312	49.68	225	348	-	-	-	-
ho_nopr	13509	17303	2.55	0.00	0.55	0.00	5271350	0.00	-	-	-	1791	101.80	325	11392	-	-	-	-
ho_noprxs	13509	17303	2.52	0.00	0.59	0.00	5159189	0.00	-	-	-	12611	31.37	897	-	-	-	-	-
ho_noxs	13509	17303	0.25	0.00	0.07	0.00	475460	0.00	0.10	12402	194	586	34.96	325	-	-	-	-	-
hybrid	13509	17303	293.55	0.00	-	-	-	-	-	-	2115	-	-	-	-	-	-	-	-
k	13509	17303	1.86	0.13	0.08	0.06	1329209	0.16	0.14	12880	188	-	-	-	-	-	-	1.36	0.31
ks	13509	17303	2.27	0.01	0.06	0.06	1006877	0.01	0.11	12880	31327	-	-	-	-	-	3003	-	-
ks_nopr	13509	17303	5.17	0.02	0.06	0.00	1257523	0.01	0.12	12880	0	-	-	-	-	-	80887	-	-
ni	13509	17303	0.30	0.00	0.07	0.00	223396	0.00	0.09	12402	743	-	-	-	-	84	-	-	-
ni_nopr	13509	17303	295.65	0.00	284.08	0.00	228561981	0.00	-	-	-	-	-	-	-	8842	-	-	-
ho	13509	17358	0.23	0.00	0.06	0.00	354901	0.00	0.12	12582	187	246	36.86	189	303	-	-	-	-
ho_nopr	13509	17358	2.75	0.00	2.63	0.00	5649156	0.00	-	-	-	1856	113.87	372	11280	-	-	-	-
ho_noprxs	13509	17358	2.83	0.00	2.68	0.00	5847172	0.00	-	-	-	12512	30.91	996	-	-	-	-	-
ho_noxs	13509	17358	0.22	0.00	0.05	0.00	373319	0.00	0.12	12582	178	488	20.24	259	-	-	-	-	-
hybrid	13509	17358	327.05	0.00	-	-	-	-	-	-	1990	-	-	-	-	-	-	-	-
k	13509	17358	1.91	0.43	0.06	0.09	1520246	0.47	0.14	12896	154	-	-	-	-	-	-	1.42	0.30
ks	13509	17358	2.35	0.01	0.04	0.11	1043378	0.01	0.12	12896	32159	-	-	-	-	-	3255	-	-
ks_nopr	13509	17358	5.52	0.01	0.05	0.11	1340760	0.01	0.12	12896	0	-	-	-	-	-	90390	-	-
ni	13509	17358	0.28	0.00	0.05	0.00	188778	0.00	0.11	12582	593	-	-	-	-	82	-	-	-
ni_nopr	13509	17358	323.10	0.00	299.13	0.00	245728170	0.00	-	-	-	-	-	-	-	9227	-	-	-
ho	1400	2231	0.04	0.00	0.03	0.00	83957	0.00	0.01	1091	38	89	50.24	54	126	-	-	-	-
ho_nopr	1400	2231	0.09	0.00	0.05	0.00	242591	0.00	-	-	-	291	123.47	110	998	-	-	-	-
ho_noprxs	1400	2231	0.15	0.00	0.07	0.00	326020	0.00	-	-	-	1170	149.67	229	-	-	-	-	-
ho_noxs	1400	2231	0.03	0.00	0.03	0.00	90377	0.00	0.01	1091	37	189	32.30	81	-	-	-	-	-
hybrid	1400	2231	1.88	0.00	-	-	-	-	-	-	240	-	-	-	-	-	-	-	-
k	1400	2231	0.17	0.35	0.11	0.54	200415	0.32	0.02	1213	78	-	-	-	-	-	-	0.09	0.06
ks	1400	2231	0.44	0.02	0.05	0.56	197380	0.02	0.01	1213	6914	-	-	-	-	-	1412	-	-
ks_nopr	1400	2231	0.92	0.07	0.16	0.78	226231	0.04	0.02	1213	0	-	-	-	-	-	15587	-	-
ni	1400	2231	0.03	0.00	0.03	0.00	32502	0.00	0.01	1091	222	-	-	-	-	26	-	-	-
ni_nopr	1400	2231	2.18	0.00	0.20	0.00	3411931	0.00	-	-	-	-	-	-	-	978	-	-	-
ho	13509	17375	0.22	0.00	0.22	0.00	329559	0.00	0.12	12714	147	175	51.47	164	308	-	-	-	-
ho_nopr	13509	17375	2.77	0.00	1.91	0.00	5755826	0.00	-	-	-	1600	142.04	293	11615	-	-	-	-
ho_noprxs	13509	17375	2.98	0.00	1.92	0.00	6142642	0.00	-	-	-	12593	35.86	915	-	-	-	-	-
ho_noxs	13509	17375	0.21	0.00	0.18	0.00	353291	0.00	0.11	12714	109	424	24.79	260	-	-	-	-	-
hybrid	13509	17375	291.32	0.00	-	-	-	-	-	-	1875	-	-	-	-	-	-	-	-
k	13509	17375	1.62	0.11	0.17	0.00	1215379	0.14	0.14	12996	100	-	-	-	-	-	-	1.19	0.25
ks	13509	17375	1.72	0.01	0.14	0.04	790518	0.01	0.12	12996	23072	-	-	-	-	-	2358	-	-
ks_nopr	13509	17375	3.43	0.01	0.14	0.03	907779	0.01	0.12	12996	0	-	-	-	-	-	48821	-	-
ni	13509	17375	0.26	0.00	0.15	0.00	168251	0.00	0.10	12714	459	-	-	-	-	66	-	-	-
ni_nopr	13509	17375	278.35	0.00	161.82	0.00	214819246	0.00	-	-	-	-	-	-	-	8342	-	-	-

Table A.4: TSP data (cont)

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	13509	17386	0.24	0.00	0.20	0.00	403433	0.00	0.11	12343	169	252	38.65	241	503	-	-	-	-
ho_nopr	13509	17386	2.43	0.00	2.16	0.00	4927526	0.00	-	-	-	1512	164.06	329	11667	-	-	-	-
ho_noprxs	13509	17386	2.44	0.00	2.15	0.00	4882956	0.00	-	-	-	12526	35.40	982	-	-	-	-	-
ho_noxs	13509	17386	0.24	0.00	0.22	0.00	405268	0.00	0.11	12343	229	585	20.25	350	-	-	-	-	-
hybrid	13509	17386	282.56	0.00	-	-	-	-	-	-	2086	-	-	-	-	-	-	-	-
k	13509	17386	2.42	0.27	0.16	0.02	1753774	0.34	0.14	12850	137	-	-	-	-	-	-	1.90	0.33
ks	13509	17386	2.13	0.01	0.13	0.00	951413	0.00	0.12	12850	29675	-	-	-	-	-	2609	-	-
ks_nopr	13509	17386	4.23	0.01	0.13	0.00	1090329	0.00	0.12	12850	0	-	-	-	-	-	56422	-	-
ni	13509	17386	0.27	0.00	0.16	0.00	173634	0.00	0.10	12343	726	-	-	-	-	54	-	-	-
ni_nopr	13509	17386	258.39	0.00	6.02	0.00	198286438	0.00	-	-	-	-	-	-	-	7792	-	-	-
ho	13509	17390	0.25	0.00	0.03	0.00	417099	0.00	0.10	12385	125	331	35.62	296	371	-	-	-	-
ho_nopr	13509	17390	2.99	0.00	0.02	0.00	6009698	0.00	-	-	-	1821	147.39	362	11325	-	-	-	-
ho_noprxs	13509	17390	3.25	0.00	0.02	0.00	6302949	0.00	-	-	-	12600	66.86	908	-	-	-	-	-
ho_noxs	13509	17390	0.25	0.00	0.03	0.00	454853	0.00	0.11	12385	147	629	20.88	346	-	-	-	-	-
hybrid	13509	17390	286.08	0.00	-	-	-	-	-	-	2435	-	-	-	-	-	-	-	-
k	13509	17390	1.60	0.27	0.02	0.18	1198111	0.25	0.14	12874	147	-	-	-	-	-	-	1.11	0.30
ks	13509	17390	2.37	0.01	0.02	0.00	1050881	0.01	0.11	12874	32502	-	-	-	-	-	3184	-	-
ks_nopr	13509	17390	5.59	0.01	0.02	0.22	1345529	0.01	0.11	12874	0	-	-	-	-	-	89900	-	-
ni	13509	17390	0.30	0.00	0.02	0.00	207357	0.00	0.09	12385	738	-	-	-	-	68	-	-	-
ni_nopr	13509	17390	267.68	0.00	0.02	0.00	210277258	0.00	-	-	-	-	-	-	-	8291	-	-	-
ho	13509	17494	0.21	0.00	0.07	0.00	321937	0.00	0.11	12626	136	196	34.49	205	345	-	-	-	-
ho_nopr	13509	17494	3.15	0.00	2.41	0.00	6597998	0.00	-	-	-	1631	106.84	296	11581	-	-	-	-
ho_noprxs	13509	17494	2.38	0.00	1.63	0.00	4950277	0.00	-	-	-	12534	31.10	974	-	-	-	-	-
ho_noxs	13509	17494	0.20	0.00	0.07	0.00	331620	0.00	0.11	12626	122	469	16.97	291	-	-	-	-	-
hybrid	13509	17494	312.60	0.00	-	-	-	-	-	-	1981	-	-	-	-	-	-	-	-
k	13509	17494	1.02	0.36	0.07	0.06	801265	0.29	0.15	12966	174	-	-	-	-	-	-	0.59	0.23
ks	13509	17494	1.88	0.01	0.05	0.09	854202	0.01	0.12	12966	25581	-	-	-	-	-	2523	-	-
ks_nopr	13509	17494	4.04	0.01	0.06	0.07	1029351	0.01	0.12	12966	0	-	-	-	-	-	62074	-	-
ni	13509	17494	0.27	0.00	0.06	0.00	189659	0.00	0.10	12626	569	-	-	-	-	76	-	-	-
ni_nopr	13509	17494	245.45	0.00	174.09	0.00	187529662	0.00	-	-	-	-	-	-	-	7450	-	-	-
ho	5934	7287	0.06	0.00	0.06	0.00	93650	0.00	0.03	5651	29	40	43.60	57	155	-	-	-	-
ho_nopr	5934	7287	0.94	0.00	0.92	0.00	2142016	0.00	-	-	-	235	581.77	93	5605	-	-	-	-
ho_noprxs	5934	7287	1.06	0.00	0.88	0.00	2335250	0.00	-	-	-	5563	74.45	370	-	-	-	-	-
ho_noxs	5934	7287	0.06	0.00	0.06	0.00	101110	0.00	0.03	5651	87	121	21.42	73	-	-	-	-	-
hybrid	5934	7287	16.47	0.00	-	-	-	-	-	-	787	-	-	-	-	-	-	-	-
k	5934	7287	0.22	0.29	0.05	0.09	216788	0.28	0.05	5762	40	-	-	-	-	-	-	0.10	0.05
ks	5934	7287	0.42	0.02	0.04	0.10	201301	0.01	0.03	5762	5412	-	-	-	-	-	1127	-	-
ks_nopr	5934	7287	0.74	0.07	0.04	0.00	213926	0.04	0.04	5762	0	-	-	-	-	-	10431	-	-
ni	5934	7287	0.06	0.00	0.05	0.00	29911	0.00	0.03	5651	169	-	-	-	-	16	-	-	-
ni_nopr	5934	7287	20.78	0.00	3.38	0.00	21075510	0.00	-	-	-	-	-	-	-	2274	-	-	-
ho	5934	7627	0.08	0.00	0.08	0.00	157268	0.00	0.04	5444	39	76	54.67	116	258	-	-	-	-
ho_nopr	5934	7627	0.86	0.00	0.86	0.00	2088813	0.00	-	-	-	228	259.64	158	5547	-	-	-	-
ho_noprxs	5934	7627	0.85	0.00	0.84	0.00	1916261	0.00	-	-	-	5551	34.15	382	-	-	-	-	-
ho_noxs	5934	7627	0.09	0.00	0.09	0.00	170349	0.00	0.04	5444	97	233	28.91	159	-	-	-	-	-
hybrid	5934	7627	69.72	0.00	-	-	-	-	-	-	1090	-	-	-	-	-	-	-	-
k	5934	7627	0.82	0.39	0.06	0.06	842276	0.43	0.05	5655	53	-	-	-	-	-	-	0.64	0.12
ks	5934	7627	1.08	0.02	0.11	0.32	469127	0.01	0.04	5655	13497	-	-	-	-	-	3512	-	-
ks_nopr	5934	7627	2.49	0.03	0.69	0.55	589774	0.02	0.04	5655	0	-	-	-	-	-	45318	-	-
ni	5934	7627	0.09	0.00	0.05	0.00	66531	0.00	0.03	5444	355	-	-	-	-	44	-	-	-
ni_nopr	5934	7627	60.32	0.00	0.99	0.00	60040418	0.00	-	-	-	-	-	-	-	4939	-	-	-
ho	1323	2169	0.03	0.00	0.01	0.00	51756	0.00	0.00	1115	30	41	37.41	56	79	-	-	-	-
ho_nopr	1323	2169	0.10	0.00	0.09	0.00	270015	0.00	-	-	-	175	75.35	72	1075	-	-	-	-
ho_noprxs	1323	2169	0.12	0.00	0.12	0.00	348029	0.00	-	-	-	1188	22.89	134	-	-	-	-	-
ho_noxs	1323	2169	0.02	0.00	0.00	0.00	54641	0.00	0.01	1115	49	93	23.39	64	-	-	-	-	-
hybrid	1323	2169	3.64	0.00	-	-	-	-	-	-	338	-	-	-	-	-	-	-	-
k	1323	2169	0.16	0.36	0.00	1.22	203805	0.39	0.01	1205	17	-	-	-	-	-	-	0.10	0.04
ks	1323	2169	0.42	0.03	0.00	1.22	177601	0.02	0.01	1205	5574	-	-	-	-	-	1900	-	-
ks_nopr	1323	2169	0.93	0.04	0.00	1.22	213233	0.02	0.01	1205	0	-	-	-	-	-	19284	-	-
ni	1323	2169	0.04	0.00	0.00	0.00	43844	0.00	0.01	1115	141	-	-	-	-	60	-	-	-
ni_nopr	1323	2169	3.83	0.00	1.94	0.00	5686288	0.00	-	-	-	-	-	-	-	1279	-	-	-

Table A.5: TSP data (cont)

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	1323	2195	0.02	0.00	0.01	0.00	48710	0.00	0.01	1167	26	39	37.49	52	38	-	-	-	-
ho_nopr	1323	2195	0.13	0.00	0.04	0.00	392661	0.00	-	-	-	148	81.22	93	1081	-	-	-	-
ho_noprxs	1323	2195	0.14	0.00	0.03	0.00	396920	0.00	-	-	-	1166	19.61	156	-	-	-	-	-
ho_noxs	1323	2195	0.02	0.00	0.01	0.00	51236	0.00	0.01	1167	29	71	23.94	54	-	-	-	-	-
hybrid	1323	2195	3.86	0.00	-	-	-	-	-	-	368	-	-	-	-	-	-	-	-
k	1323	2195	0.10	0.38	0.07	0.56	130127	0.40	0.01	1214	22	-	-	-	-	-	-	0.05	0.04
ks	1323	2195	0.41	0.03	0.03	0.46	173774	0.01	0.01	1214	5353	-	-	-	-	-	1910	-	-
ks_nopr	1323	2195	0.85	0.04	0.20	0.57	204395	0.03	0.01	1214	0	-	-	-	-	-	19234	-	-
ni	1323	2195	0.03	0.00	0.03	0.00	32044	0.00	0.01	1167	104	-	-	-	-	44	-	-	-
ni_nopr	1323	2195	4.07	0.00	3.98	0.00	5885243	0.00	-	-	-	-	-	-	-	1293	-	-	-
ho	1084	1252	0.01	0.00	0.00	0.00	11584	0.00	0.00	1052	7	7	15.57	5	11	-	-	-	-
ho_nopr	1084	1252	0.05	0.00	0.00	0.00	137721	0.00	-	-	-	90	163.66	8	985	-	-	-	-
ho_noprxs	1084	1252	0.09	0.00	0.00	0.00	211424	0.00	-	-	-	1037	122.68	46	-	-	-	-	-
ho_noxs	1084	1252	0.01	0.00	0.00	0.00	11744	0.00	0.01	1052	14	12	12.75	4	-	-	-	-	-
hybrid	1084	1252	1.36	0.00	-	-	-	-	-	-	336	-	-	-	-	-	-	-	-
k	1084	1252	0.01	0.00	0.00	0.00	13939	0.02	0.00	1061	13	-	-	-	-	-	-	0.00	0.00
ks	1084	1252	0.03	0.30	0.00	0.00	18665	0.05	0.00	1061	519	-	-	-	-	-	359	-	-
ks_nopr	1084	1252	0.09	0.16	0.00	0.00	23967	0.05	0.00	1061	0	-	-	-	-	-	2452	-	-
ni	1084	1252	0.01	0.00	0.00	0.00	3759	0.00	0.01	1052	19	-	-	-	-	8	-	-	-
ni_nopr	1084	1252	1.54	0.00	0.00	0.00	2341930	0.00	-	-	-	-	-	-	-	1003	-	-	-

Table A.6: TSP data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
hybrid	605	1162	0.23	0.00	-	-	-	-	-	-	15	-	-	-	-	-	-	-	-
ni	605	1162	0.11	0.00	0.00	0.00	136501	0.00	0.00	0	300	-	-	-	-	72	-	-	-
ni_nopr	605	1162	0.21	0.00	0.00	0.00	322571	0.00	-	-	-	-	-	-	-	151	-	-	-
hybrid	549	1072	0.28	0.00	-	-	-	-	-	-	12	-	-	-	-	-	-	-	-
ni	549	1072	0.10	0.00	0.08	0.00	116474	0.00	0.01	0	277	-	-	-	-	68	-	-	-
ni_nopr	549	1072	0.22	0.00	0.16	0.00	333669	0.00	-	-	-	-	-	-	-	185	-	-	-
hybrid	465	926	0.20	0.00	-	-	-	-	-	-	17	-	-	-	-	-	-	-	-
ni	465	926	0.08	0.00	0.00	0.00	100514	0.00	0.00	0	228	-	-	-	-	62	-	-	-
ni_nopr	465	926	0.16	0.00	0.00	0.00	260207	0.00	-	-	-	-	-	-	-	157	-	-	-
hybrid	542	1064	0.23	0.00	-	-	-	-	-	-	28	-	-	-	-	-	-	-	-
ni	542	1064	0.08	0.00	0.00	0.00	95845	0.00	0.00	0	274	-	-	-	-	58	-	-	-
ni_nopr	542	1064	0.15	0.00	0.00	0.00	240401	0.00	-	-	-	-	-	-	-	136	-	-	-
hybrid	573	1104	0.29	0.00	-	-	-	-	-	-	21	-	-	-	-	-	-	-	-
ni	573	1104	0.11	0.00	0.00	0.00	127352	0.00	0.00	0	299	-	-	-	-	70	-	-	-
ni_nopr	573	1104	0.25	0.00	0.00	0.00	386232	0.00	-	-	-	-	-	-	-	207	-	-	-
hybrid	476	945	0.20	0.00	-	-	-	-	-	-	17	-	-	-	-	-	-	-	-
ni	476	945	0.08	0.00	0.00	0.00	99407	0.00	0.00	0	217	-	-	-	-	58	-	-	-
ni_nopr	476	945	0.21	0.00	0.00	0.00	319209	0.00	-	-	-	-	-	-	-	191	-	-	-
hybrid	607	1150	0.22	0.00	-	-	-	-	-	-	14	-	-	-	-	-	-	-	-
ni	607	1150	0.10	0.00	0.00	0.00	110719	0.00	0.00	0	294	-	-	-	-	56	-	-	-
ni_nopr	607	1150	0.15	0.00	0.00	0.00	223849	0.00	-	-	-	-	-	-	-	112	-	-	-
hybrid	557	1091	0.27	0.00	-	-	-	-	-	-	14	-	-	-	-	-	-	-	-
ni	557	1091	0.09	0.00	0.00	0.00	113798	0.00	0.00	0	302	-	-	-	-	58	-	-	-
ni_nopr	557	1091	0.21	0.00	0.00	0.00	308862	0.00	-	-	-	-	-	-	-	162	-	-	-
hybrid	505	971	0.23	0.00	-	-	-	-	-	-	10	-	-	-	-	-	-	-	-
ni	505	971	0.09	0.00	0.00	0.00	100608	0.00	0.01	0	254	-	-	-	-	60	-	-	-
ni_nopr	505	971	0.18	0.00	0.00	0.00	278074	0.00	-	-	-	-	-	-	-	159	-	-	-
hybrid	325	561	0.03	0.00	-	-	-	-	-	-	8	-	-	-	-	-	-	-	-
ni	325	561	0.01	0.00	0.00	0.00	16351	0.00	0.00	0	126	-	-	-	-	16	-	-	-
ni_nopr	325	561	0.02	0.00	0.00	0.00	28436	0.00	-	-	-	-	-	-	-	37	-	-	-
hybrid	477	920	0.21	0.00	-	-	-	-	-	-	26	-	-	-	-	-	-	-	-
ni	477	920	0.08	0.00	0.02	0.00	86054	0.00	0.01	0	232	-	-	-	-	52	-	-	-
ni_nopr	477	920	0.15	0.00	0.02	0.00	232635	0.00	-	-	-	-	-	-	-	144	-	-	-
hybrid	449	861	0.19	0.00	-	-	-	-	-	-	19	-	-	-	-	-	-	-	-
ni	449	861	0.07	0.00	0.03	0.00	87231	0.00	0.00	0	208	-	-	-	-	56	-	-	-
ni_nopr	449	861	0.13	0.00	0.06	0.00	202259	0.00	-	-	-	-	-	-	-	130	-	-	-
hybrid	454	886	0.21	0.00	-	-	-	-	-	-	11	-	-	-	-	-	-	-	-
ni	454	886	0.08	0.00	0.00	0.00	91044	0.00	0.00	0	227	-	-	-	-	62	-	-	-
ni_nopr	454	886	0.17	0.00	0.00	0.00	264027	0.00	-	-	-	-	-	-	-	165	-	-	-
hybrid	403	786	0.14	0.00	-	-	-	-	-	-	12	-	-	-	-	-	-	-	-
ni	403	786	0.06	0.00	0.06	0.00	76546	0.00	0.00	0	207	-	-	-	-	58	-	-	-
ni_nopr	403	786	0.12	0.00	0.12	0.00	184082	0.00	-	-	-	-	-	-	-	143	-	-	-
hybrid	532	1029	0.27	0.00	-	-	-	-	-	-	28	-	-	-	-	-	-	-	-
ni	532	1029	0.08	0.00	0.00	0.00	96779	0.00	0.00	0	267	-	-	-	-	54	-	-	-
ni_nopr	532	1029	0.17	0.00	0.00	0.00	260179	0.00	-	-	-	-	-	-	-	164	-	-	-
hybrid	501	950	0.20	0.00	-	-	-	-	-	-	16	-	-	-	-	-	-	-	-
ni	501	950	0.12	0.00	0.00	0.00	140584	0.00	0.00	0	256	-	-	-	-	76	-	-	-
ni_nopr	501	950	0.24	0.00	0.00	0.00	369383	0.00	-	-	-	-	-	-	-	202	-	-	-
hybrid	492	946	0.25	0.00	-	-	-	-	-	-	19	-	-	-	-	-	-	-	-
ni	492	946	0.08	0.00	0.00	0.00	88342	0.00	0.00	0	252	-	-	-	-	52	-	-	-
ni_nopr	492	946	0.14	0.00	0.00	0.00	208264	0.00	-	-	-	-	-	-	-	122	-	-	-

Table A.7: PRETSP data

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	300	22425	0.17	0.07	0.02	0.32	414336	0.06	0.03	0	153	4	284.65	0	139	-	-	-	-
ho_nopr	300	22425	0.13	0.08	0.02	0.50	413343	0.09	-	-	-	8	276.59	16	274	-	-	-	-
ho_noprxs	300	22425	0.19	0.04	0.02	0.31	750026	0.05	-	-	-	49	187.19	249	-	-	-	-	-
ho_noxs	300	22425	0.16	0.06	0.02	0.32	418143	0.06	0.03	0	281	8	270.49	7	-	-	-	-	-
hybrid	300	17632	0.21	0.05	-	-	-	-	-	-	3	-	-	-	-	-	-	-	-
k	300	22425	0.33	0.03	0.03	0.32	334714	0.01	0.03	0	330	-	-	-	-	-	-	-	-
ks	300	22425	7.31	0.05	0.02	0.37	1978048	0.04	0.03	0	5775	-	-	-	-	-	265	-	-
ks_nopr	300	22425	7.35	0.04	0.02	0.45	1870356	0.03	0.03	0	0	-	-	-	-	-	5295	-	-
ni	300	22425	0.14	0.03	0.02	0.34	150473	0.03	0.03	0	175	-	-	-	-	2	-	-	-
ni_nopr	300	22425	0.12	0.11	0.02	0.50	185572	0.08	-	-	-	-	-	-	-	4	-	-	-
ho	400	39900	0.32	0.04	0.03	0.30	758316	0.05	0.05	0	233	5	396.31	0	159	-	-	-	-
ho_nopr	400	39900	0.28	0.09	0.02	0.34	858315	0.08	-	-	-	12	374.57	37	349	-	-	-	-
ho_noprxs	400	39900	0.42	0.10	0.02	0.42	1553980	0.10	-	-	-	53	235.00	345	-	-	-	-	-
ho_noxs	400	39900	0.30	0.03	0.03	0.12	724352	0.06	0.05	0	384	8	350.95	5	-	-	-	-	-
hybrid	400	31413	0.42	0.05	-	-	-	-	-	-	3	-	-	-	-	-	-	-	-
k	400	39900	0.62	0.02	0.05	0.31	595073	0.01	0.05	0	454	-	-	-	-	-	-	-	-
ks	400	39900	14.94	0.03	0.03	0.34	3724151	0.03	0.05	0	7960	-	-	-	-	-	282	-	-
ks_nopr	400	39900	14.07	0.03	0.03	0.37	3460471	0.03	0.05	0	0	-	-	-	-	-	7059	-	-
ni	400	39900	0.25	0.03	0.03	0.36	269833	0.03	0.05	0	243	-	-	-	-	2	-	-	-
ni_nopr	400	39900	0.24	0.07	0.03	0.40	339220	0.08	-	-	-	-	-	-	-	4	-	-	-
ho	500	62375	0.49	0.06	0.03	0.64	1123700	0.07	0.08	0	313	4	496.79	0	180	-	-	-	-
ho_nopr	500	62375	0.45	0.08	0.02	0.67	1395214	0.08	-	-	-	13	456.54	41	444	-	-	-	-
ho_noprxs	500	62375	0.67	0.05	0.02	0.78	2374071	0.04	-	-	-	70	298.57	429	-	-	-	-	-
ho_noxs	500	62375	0.48	0.05	0.03	0.54	1126768	0.07	0.08	0	482	7	439.17	8	-	-	-	-	-
hybrid	500	49094	0.69	0.07	-	-	-	-	-	-	4	-	-	-	-	-	-	-	-
k	500	62375	1.00	0.01	0.05	0.61	928399	0.01	0.08	0	571	-	-	-	-	-	-	-	-
ks	500	62375	25.04	0.06	0.03	0.64	5980270	0.05	0.08	0	10323	-	-	-	-	-	288	-	-
ks_nopr	500	62375	23.14	0.06	0.03	0.62	5556688	0.05	0.07	0	0	-	-	-	-	-	8877	-	-
ni	500	62375	0.42	0.04	0.03	0.64	420353	0.04	0.07	0	306	-	-	-	-	2	-	-	-
ni_nopr	500	62375	0.38	0.09	0.03	0.60	536465	0.12	-	-	-	-	-	-	-	5	-	-	-
ho	600	89850	0.69	0.01	0.08	0.51	1525698	0.02	0.11	0	406	3	598.40	0	187	-	-	-	-
ho_nopr	600	89850	0.70	0.06	0.06	0.55	2132219	0.06	-	-	-	13	558.40	69	516	-	-	-	-
ho_noprxs	600	89850	1.09	0.07	0.06	0.48	3828988	0.07	-	-	-	77	318.65	521	-	-	-	-	-
ho_noxs	600	89850	0.68	0.02	0.08	0.46	1535212	0.02	0.11	0	560	5	497.57	32	-	-	-	-	-
hybrid	600	70635	1.04	0.06	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-
k	600	89850	1.49	0.01	0.13	0.46	1344394	0.01	0.12	0	713	-	-	-	-	-	-	-	-
ks	600	89850	38.91	0.02	0.08	0.51	9118613	0.02	0.11	0	13265	-	-	-	-	-	299	-	-
ks_nopr	600	89850	35.65	0.03	0.08	0.49	8449323	0.03	0.11	0	0	-	-	-	-	-	12059	-	-
ni	600	89850	0.63	0.02	0.08	0.46	620847	0.02	0.10	0	404	-	-	-	-	2	-	-	-
ni_nopr	600	89850	0.61	0.08	0.08	0.50	837674	0.09	-	-	-	-	-	-	-	5	-	-	-
ho	700	122325	1.01	0.06	0.13	0.26	2255858	0.08	0.15	0	499	4	697.77	0	194	-	-	-	-
ho_nopr	700	122325	1.11	0.07	0.10	0.25	3361271	0.06	-	-	-	13	644.92	71	615	-	-	-	-
ho_noprxs	700	122325	1.44	0.04	0.09	0.31	5029798	0.06	-	-	-	95	412.79	603	-	-	-	-	-
ho_noxs	700	122325	0.99	0.05	0.12	0.29	2260789	0.08	0.15	0	656	6	592.04	35	-	-	-	-	-
hybrid	700	96252	1.60	0.06	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-
k	700	122325	2.09	0.01	0.20	0.23	1850405	0.01	0.16	0	873	-	-	-	-	-	-	-	-
ks	700	122325	56.29	0.02	0.12	0.27	13160519	0.02	0.15	0	16922	-	-	-	-	-	305	-	-
ks_nopr	700	122325	52.35	0.02	0.12	0.26	12319490	0.02	0.15	0	0	-	-	-	-	-	17121	-	-
ni	700	122325	0.90	0.02	0.13	0.26	880236	0.02	0.15	0	530	-	-	-	-	2	-	-	-
ni_nopr	700	122325	0.97	0.08	0.12	0.26	1347265	0.10	-	-	-	-	-	-	-	6	-	-	-
ho	800	159800	1.38	0.04	0.12	0.72	3030685	0.07	0.20	0	561	4	796.93	0	232	-	-	-	-
ho_nopr	800	159800	1.37	0.04	0.09	0.77	4194031	0.04	-	-	-	15	729.50	99	683	-	-	-	-
ho_noprxs	800	159800	1.88	0.03	0.09	0.73	6540903	0.04	-	-	-	106	464.84	693	-	-	-	-	-
ho_noxs	800	159800	1.36	0.04	0.11	0.75	3043199	0.07	0.20	0	772	8	699.15	17	-	-	-	-	-
hybrid	800	125750	2.00	0.07	-	-	-	-	-	-	7	-	-	-	-	-	-	-	-
k	800	159800	2.75	0.01	0.18	0.73	2396977	0.01	0.21	0	971	-	-	-	-	-	-	-	-
ks	800	159800	73.57	0.03	0.11	0.75	17143931	0.04	0.19	0	18982	-	-	-	-	-	311	-	-
ks_nopr	800	159800	67.76	0.04	0.11	0.73	15878903	0.04	0.19	0	0	-	-	-	-	-	17362	-	-
ni	800	159800	1.14	0.03	0.11	0.75	1116805	0.04	0.19	0	560	-	-	-	-	2	-	-	-
ni_nopr	800	159800	1.18	0.11	0.11	0.71	1583954	0.13	-	-	-	-	-	-	-	5	-	-	-

Table A.8: NOI1 data

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	900	202275	1.71	0.04	0.16	0.67	3612659	0.06	0.25	0	661	4	897.87	0	232	-	-	-	-
ho_nopr	900	202275	1.86	0.03	0.11	0.71	5794794	0.04	-	-	-	18	831.14	187	692	-	-	-	-
ho_noprxs	900	202275	2.51	0.03	0.11	0.72	8788232	0.04	-	-	-	96	495.86	803	-	-	-	-	-
ho_noxs	900	202275	1.66	0.04	0.15	0.69	3644518	0.06	0.25	0	855	6	766.31	35	-	-	-	-	-
hybrid	900	159134	2.72	0.06	-	-	-	-	-	-	7	-	-	-	-	-	-	-	-
k	900	202275	3.56	0.01	0.24	0.67	3057606	0.01	0.27	0	1133	-	-	-	-	-	-	-	-
ks	900	202275	95.74	0.02	0.15	0.68	22306145	0.02	0.25	0	22424	-	-	-	-	-	312	-	-
ks_nopr	900	202275	89.02	0.02	0.15	0.68	20755059	0.02	0.25	0	0	-	-	-	-	-	21934	-	-
ni	900	202275	1.52	0.03	0.15	0.66	1458977	0.02	0.25	0	683	-	-	-	-	2	-	-	-
ni_nopr	900	202275	1.65	0.07	0.15	0.69	2243445	0.09	-	-	-	-	-	-	-	6	-	-	-
ho	1000	249750	2.13	0.02	0.17	0.57	4359927	0.01	0.31	0	754	4	997.52	0	238	-	-	-	-
ho_nopr	1000	249750	2.46	0.09	0.12	0.65	7605133	0.09	-	-	-	16	933.95	243	738	-	-	-	-
ho_noprxs	1000	249750	3.22	0.13	0.12	0.63	11050154	0.14	-	-	-	100	566.37	898	-	-	-	-	-
ho_noxs	1000	249750	2.07	0.01	0.16	0.58	4412605	0.02	0.31	0	964	7	854.90	26	-	-	-	-	-
hybrid	1000	196545	3.54	0.03	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-
k	1000	249750	4.44	0.01	0.26	0.57	3786958	0.00	0.34	0	1280	-	-	-	-	-	-	-	-
ks	1000	249750	122.10	0.01	0.16	0.60	28411005	0.01	0.31	0	25977	-	-	-	-	-	318	-	-
ks_nopr	1000	249750	115.18	0.01	0.15	0.59	26625662	0.01	0.30	0	0	-	-	-	-	-	26448	-	-
ni	1000	249750	1.89	0.01	0.16	0.59	1822543	0.01	0.30	0	787	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	2.16	0.04	0.15	0.60	2941612	0.05	-	-	-	-	-	-	-	7	-	-	-

Table A.9: NOI1 data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	300	22425	0.14	0.10	0.12	0.14	317385	0.12	0.03	0	65	5	210.41	0	227	-	-	-	-
ho_nopr	300	22425	0.11	0.04	0.09	0.05	349113	0.05	-	-	-	10	178.96	0	288	-	-	-	-
ho_noprxs	300	22425	0.18	0.10	0.09	0.18	717747	0.11	-	-	-	83	106.86	215	-	-	-	-	-
ho_noxs	300	22425	0.17	0.15	0.11	0.15	417408	0.19	0.03	0	193	35	147.06	69	-	-	-	-	-
hybrid	300	17632	0.19	0.04	-	-	-	-	-	-	13	-	-	-	-	-	-	-	-
k	300	22425	0.31	0.03	0.11	0.09	312624	0.00	0.03	0	203	-	-	-	-	-	-	-	-
ks	300	22425	4.39	0.02	0.32	0.42	1105148	0.02	0.03	0	4878	-	-	-	-	-	142	-	-
ks_nopr	300	22425	4.90	0.03	0.52	0.65	1257706	0.03	0.03	0	0	-	-	-	-	-	1445	-	-
ni	300	22425	0.11	0.04	0.06	0.00	121916	0.03	0.03	0	15	-	-	-	-	2	-	-	-
ni_nopr	300	22425	0.09	0.05	0.04	0.11	122219	0.03	-	-	-	-	-	-	-	2	-	-	-
ho	400	39900	0.27	0.14	0.23	0.17	563308	0.17	0.05	0	87	5	289.46	3	302	-	-	-	-
ho_nopr	400	39900	0.21	0.07	0.17	0.06	632333	0.06	-	-	-	8	244.24	6	384	-	-	-	-
ho_noprxs	400	39900	0.35	0.04	0.17	0.06	1269434	0.07	-	-	-	114	138.62	284	-	-	-	-	-
ho_noxs	400	39900	0.32	0.19	0.22	0.10	757283	0.19	0.05	0	264	44	167.76	89	-	-	-	-	-
hybrid	400	31413	0.37	0.02	-	-	-	-	-	-	22	-	-	-	-	-	-	-	-
k	400	39900	0.60	0.02	0.21	0.05	556486	0.00	0.06	0	286	-	-	-	-	-	-	-	-
ks	400	39900	8.31	0.02	0.63	0.64	2002081	0.02	0.05	0	6579	-	-	-	-	-	147	-	-
ks_nopr	400	39900	9.12	0.03	0.97	0.74	2265468	0.02	0.05	0	0	-	-	-	-	-	1675	-	-
ni	400	39900	0.22	0.03	0.12	0.03	219118	0.02	0.05	0	27	-	-	-	-	2	-	-	-
ni_nopr	400	39900	0.17	0.02	0.07	0.00	219691	0.02	-	-	-	-	-	-	-	2	-	-	-
ho	500	62375	0.49	0.13	0.33	0.07	1046260	0.17	0.08	0	108	5	360.57	0	384	-	-	-	-
ho_nopr	500	62375	0.38	0.05	0.25	0.08	1075556	0.05	-	-	-	11	271.56	0	487	-	-	-	-
ho_noprxs	500	62375	0.54	0.05	0.23	0.09	1915919	0.06	-	-	-	138	171.03	360	-	-	-	-	-
ho_noxs	500	62375	0.60	0.14	0.31	0.03	1395957	0.17	0.08	0	342	67	211.45	88	-	-	-	-	-
hybrid	500	49094	0.60	0.03	-	-	-	-	-	-	35	-	-	-	-	-	-	-	-
k	500	62375	0.97	0.02	0.33	0.05	868986	0.00	0.08	0	372	-	-	-	-	-	-	-	-
ks	500	62375	13.39	0.03	1.08	0.66	3180333	0.03	0.08	0	8341	-	-	-	-	-	150	-	-
ks_nopr	500	62375	14.54	0.02	1.29	0.67	3575272	0.02	0.07	0	0	-	-	-	-	-	1869	-	-
ni	500	62375	0.36	0.02	0.19	0.00	348878	0.02	0.08	0	40	-	-	-	-	2	-	-	-
ni_nopr	500	62375	0.29	0.01	0.11	0.04	351416	0.03	-	-	-	-	-	-	-	2	-	-	-
ho	600	89850	0.75	0.08	0.54	0.16	1560962	0.10	0.11	0	142	5	539.13	0	449	-	-	-	-
ho_nopr	600	89850	0.57	0.03	0.38	0.09	1614625	0.01	-	-	-	15	306.51	0	583	-	-	-	-
ho_noprxs	600	89850	0.80	0.04	0.35	0.12	2729178	0.04	-	-	-	188	200.65	410	-	-	-	-	-
ho_noxs	600	89850	0.94	0.05	0.49	0.15	2139332	0.06	0.11	0	458	67	249.82	71	-	-	-	-	-
hybrid	600	70635	0.88	0.01	-	-	-	-	-	-	28	-	-	-	-	-	-	-	-
k	600	89850	1.43	0.01	0.49	0.03	1249364	0.00	0.12	0	449	-	-	-	-	-	-	-	-
ks	600	89850	19.71	0.02	1.38	0.51	4609111	0.02	0.11	0	10117	-	-	-	-	-	153	-	-
ks_nopr	600	89850	21.33	0.02	2.13	0.64	5217585	0.02	0.11	0	0	-	-	-	-	-	2110	-	-
ni	600	89850	0.53	0.01	0.27	0.02	496450	0.01	0.11	0	46	-	-	-	-	2	-	-	-
ni_nopr	600	89850	0.42	0.01	0.17	0.03	498021	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	700	122325	1.00	0.05	0.76	0.16	2011223	0.05	0.15	0	227	1	699.70	0	469	-	-	-	-
ho_nopr	700	122325	0.81	0.02	0.53	0.07	2240568	0.02	-	-	-	16	357.63	0	682	-	-	-	-
ho_noprxs	700	122325	1.16	0.04	0.50	0.09	3881323	0.05	-	-	-	219	232.98	479	-	-	-	-	-
ho_noxs	700	122325	1.28	0.07	0.68	0.11	2869285	0.07	0.15	0	547	73	334.32	76	-	-	-	-	-
hybrid	700	96252	1.25	0.02	-	-	-	-	-	-	71	-	-	-	-	-	-	-	-
k	700	122325	1.99	0.01	0.67	0.03	1705377	0.00	0.16	0	546	-	-	-	-	-	-	-	-
ks	700	122325	27.83	0.03	2.39	0.72	6488656	0.03	0.15	0	12090	-	-	-	-	-	159	-	-
ks_nopr	700	122325	29.94	0.03	2.60	0.64	7256479	0.02	0.15	0	0	-	-	-	-	-	2346	-	-
ni	700	122325	0.73	0.01	0.37	0.01	688362	0.01	0.15	0	81	-	-	-	-	2	-	-	-
ni_nopr	700	122325	0.59	0.01	0.23	0.02	692674	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	800	159800	1.51	0.07	1.16	0.19	3096715	0.09	0.20	0	188	4	710.30	3	602	-	-	-	-
ho_nopr	800	159800	1.10	0.04	0.78	0.14	3205484	0.06	-	-	-	15	449.75	3	780	-	-	-	-
ho_noprxs	800	159800	1.62	0.04	0.73	0.12	5505613	0.05	-	-	-	225	268.59	573	-	-	-	-	-
ho_noxs	800	159800	1.80	0.03	1.04	0.13	4039553	0.04	0.19	0	603	100	335.84	94	-	-	-	-	-
hybrid	800	125750	1.65	0.01	-	-	-	-	-	-	90	-	-	-	-	-	-	-	-
k	800	159800	2.64	0.01	0.87	0.02	2225934	0.00	0.21	0	628	-	-	-	-	-	-	-	-
ks	800	159800	37.28	0.03	3.24	0.68	8607079	0.03	0.19	0	13977	-	-	-	-	-	162	-	-
ks_nopr	800	159800	39.85	0.02	3.50	0.60	9606802	0.02	0.19	0	0	-	-	-	-	-	2506	-	-
ni	800	159800	0.98	0.01	0.49	0.01	904395	0.00	0.19	0	107	-	-	-	-	2	-	-	-
ni_nopr	800	159800	0.78	0.01	0.30	0.02	911602	0.01	-	-	-	-	-	-	-	2	-	-	-

Table A.10: NOI2 data

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	900	202275	1.77	0.09	1.26	0.13	3460314	0.11	0.25	0	276	3	831.16	0	618	-	-	-	-
ho_nopr	900	202275	1.42	0.02	0.90	0.06	3908538	0.01	-	-	-	17	459.41	0	881	-	-	-	-
ho_noprxs	900	202275	2.00	0.02	0.84	0.03	6646862	0.02	-	-	-	270	294.70	628	-	-	-	-	-
ho_noxs	900	202275	2.35	0.03	1.17	0.08	5118598	0.03	0.25	0	650	126	385.05	121	-	-	-	-	-
hybrid	900	159134	2.14	0.02	-	-	-	-	-	-	115	-	-	-	-	-	-	-	-
k	900	202275	3.41	0.01	1.14	0.03	2819478	0.00	0.27	0	721	-	-	-	-	-	-	-	-
ks	900	202275	47.91	0.02	4.97	0.45	11034605	0.02	0.24	0	15924	-	-	-	-	-	163	-	-
ks_nopr	900	202275	50.74	0.02	3.76	0.60	12171827	0.02	0.25	0	0	-	-	-	-	-	2654	-	-
ni	900	202275	1.25	0.01	0.62	0.01	1155836	0.01	0.24	0	135	-	-	-	-	2	-	-	-
ni_nopr	900	202275	1.01	0.02	0.38	0.00	1168331	0.02	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	2.59	0.10	1.65	0.18	5197277	0.12	0.30	0	177	11	681.56	0	810	-	-	-	-
ho_nopr	1000	249750	1.81	0.02	1.15	0.11	5145474	0.03	-	-	-	17	530.75	0	982	-	-	-	-
ho_noprxs	1000	249750	2.49	0.04	1.09	0.06	8270216	0.04	-	-	-	315	322.80	684	-	-	-	-	-
ho_noxs	1000	249750	2.92	0.03	1.56	0.11	6257279	0.02	0.30	0	752	128	403.56	117	-	-	-	-	-
hybrid	1000	196545	2.70	0.02	-	-	-	-	-	-	132	-	-	-	-	-	-	-	-
k	1000	249750	4.24	0.01	1.40	0.02	3485807	0.00	0.34	0	813	-	-	-	-	-	-	-	-
ks	1000	249750	59.87	0.03	4.44	0.70	13681153	0.03	0.31	0	17980	-	-	-	-	-	164	-	-
ks_nopr	1000	249750	63.89	0.02	4.48	0.60	15250038	0.02	0.31	0	0	-	-	-	-	-	2881	-	-
ni	1000	249750	1.55	0.01	0.78	0.01	1429950	0.01	0.31	0	156	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.25	0.01	0.47	0.03	1451158	0.01	-	-	-	-	-	-	-	2	-	-	-

Table A.11: NOI2 data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	1000	24975	0.21	0.09	0.01	0.97	531168	0.10	0.03	0	262	6	989.70	0	728	-	-	-	-
ho_nopr	1000	24975	0.14	0.05	0.01	1.33	462470	0.03	-	-	-	11	969.33	0	987	-	-	-	-
ho_noprxs	1000	24975	0.39	0.03	0.01	0.94	1528034	0.03	-	-	-	73	693.05	925	-	-	-	-	-
ho_noxs	1000	24975	0.24	0.09	0.01	0.62	670385	0.08	0.03	0	709	23	920.57	265	-	-	-	-	-
hybrid	1000	24362	0.48	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	24975	0.43	0.02	0.01	0.69	439631	0.01	0.04	0	635	-	-	-	-	-	-	-	-
ks	1000	24975	7.32	0.05	0.01	0.74	1940860	0.07	0.04	0	19632	-	-	-	-	-	160	-	-
ks_nopr	1000	24975	9.12	0.08	0.01	0.69	2348652	0.07	0.03	0	0	-	-	-	-	-	4175	-	-
ni	1000	24975	0.16	0.03	0.01	0.89	166771	0.06	0.03	0	0	-	-	-	-	1	-	-	-
ni_nopr	1000	24975	0.12	0.06	0.01	1.22	166771	0.06	-	-	-	-	-	-	-	1	-	-	-
ho	1000	49950	0.42	0.05	0.04	0.23	1044323	0.06	0.06	0	462	5	980.19	10	520	-	-	-	-
ho_nopr	1000	49950	0.34	0.04	0.03	0.12	1112500	0.05	-	-	-	13	964.39	6	979	-	-	-	-
ho_noprxs	1000	49950	0.73	0.11	0.03	0.12	2803815	0.13	-	-	-	78	759.21	920	-	-	-	-	-
ho_noxs	1000	49950	0.52	0.06	0.05	0.08	1389832	0.05	0.07	0	933	31	953.76	33	-	-	-	-	-
hybrid	1000	47529	0.81	0.03	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-
k	1000	49950	0.90	0.02	0.06	0.08	872265	0.01	0.08	0	910	-	-	-	-	-	-	-	-
ks	1000	49950	22.80	0.12	0.04	0.11	5734117	0.10	0.07	0	19202	-	-	-	-	-	257	-	-
ks_nopr	1000	49950	23.20	0.06	0.05	0.15	5694113	0.05	0.07	0	0	-	-	-	-	-	8125	-	-
ni	1000	49950	0.34	0.02	0.05	0.11	356988	0.03	0.07	0	307	-	-	-	-	2	-	-	-
ni_nopr	1000	49950	0.28	0.05	0.04	0.11	383134	0.06	-	-	-	-	-	-	-	3	-	-	-
ho	1000	124875	1.35	0.14	0.06	0.66	3185443	0.14	0.16	0	570	8	986.71	0	418	-	-	-	-
ho_nopr	1000	124875	1.11	0.05	0.04	0.64	3512990	0.06	-	-	-	15	934.97	31	952	-	-	-	-
ho_noprxs	1000	124875	1.77	0.07	0.04	0.64	6382915	0.07	-	-	-	91	605.64	907	-	-	-	-	-
ho_noxs	1000	124875	1.34	0.08	0.06	0.65	3333379	0.11	0.16	0	920	16	952.81	61	-	-	-	-	-
hybrid	1000	110503	1.79	0.02	-	-	-	-	-	-	4	-	-	-	-	-	-	-	-
k	1000	124875	2.28	0.01	0.09	0.63	2077760	0.00	0.18	0	1155	-	-	-	-	-	-	-	-
ks	1000	124875	68.63	0.01	0.06	0.67	16210678	0.01	0.16	0	22233	-	-	-	-	-	317	-	-
ks_nopr	1000	124875	62.04	0.02	0.06	0.66	14661612	0.02	0.16	0	0	-	-	-	-	-	16821	-	-
ni	1000	124875	0.93	0.01	0.06	0.66	935655	0.01	0.16	0	621	-	-	-	-	2	-	-	-
ni_nopr	1000	124875	0.88	0.02	0.06	0.65	1186902	0.04	-	-	-	-	-	-	-	4	-	-	-
ho	1000	249750	2.12	0.01	0.17	0.60	4359927	0.01	0.31	0	754	4	997.52	0	238	-	-	-	-
ho_nopr	1000	249750	2.48	0.08	0.12	0.61	7605133	0.09	-	-	-	16	933.95	243	738	-	-	-	-
ho_noprxs	1000	249750	3.22	0.13	0.11	0.60	11050154	0.14	-	-	-	100	566.37	898	-	-	-	-	-
ho_noxs	1000	249750	2.07	0.01	0.16	0.58	4412605	0.02	0.31	0	964	7	854.90	26	-	-	-	-	-
hybrid	1000	196545	3.55	0.02	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-
k	1000	249750	4.41	0.01	0.25	0.57	3786958	0.00	0.34	0	1280	-	-	-	-	-	-	-	-
ks	1000	249750	121.48	0.01	0.16	0.61	28411005	0.01	0.31	0	25977	-	-	-	-	-	318	-	-
ks_nopr	1000	249750	115.19	0.01	0.15	0.58	26625662	0.01	0.31	0	0	-	-	-	-	-	26448	-	-
ni	1000	249750	1.89	0.02	0.15	0.61	1822543	0.01	0.30	0	787	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	2.16	0.04	0.16	0.59	2941612	0.05	-	-	-	-	-	-	-	7	-	-	-
ho	1000	374625	3.40	0.01	0.39	0.30	6880160	0.01	0.44	0	793	5	997.02	0	199	-	-	-	-
ho_nopr	1000	374625	3.66	0.08	0.27	0.34	10825518	0.09	-	-	-	18	906.84	215	765	-	-	-	-
ho_noprxs	1000	374625	4.37	0.08	0.27	0.34	14445572	0.09	-	-	-	124	505.85	874	-	-	-	-	-
ho_noxs	1000	374625	3.29	0.01	0.38	0.31	6856776	0.01	0.44	0	989	8	908.21	0	-	-	-	-	-
hybrid	1000	263567	5.17	0.02	-	-	-	-	-	-	6	-	-	-	-	-	-	-	-
k	1000	374625	6.32	0.01	0.63	0.29	5160278	0.00	0.47	0	1328	-	-	-	-	-	-	-	-
ks	1000	374625	159.41	0.00	0.36	0.31	37081283	0.01	0.44	0	27710	-	-	-	-	-	318	-	-
ks_nopr	1000	374625	154.62	0.01	0.36	0.31	35419759	0.01	0.44	0	0	-	-	-	-	-	31912	-	-
ni	1000	374625	2.76	0.01	0.36	0.29	2552939	0.01	0.44	0	848	-	-	-	-	2	-	-	-
ni_nopr	1000	374625	3.55	0.04	0.37	0.32	4837419	0.05	-	-	-	-	-	-	-	8	-	-	-
ho	1000	499500	4.42	0.01	0.23	0.66	8340295	0.01	0.55	0	818	5	997.06	0	174	-	-	-	-
ho_nopr	1000	499500	5.03	0.05	0.15	0.69	14703068	0.06	-	-	-	19	896.70	313	665	-	-	-	-
ho_noprxs	1000	499500	5.50	0.02	0.15	0.71	17811484	0.04	-	-	-	107	473.36	891	-	-	-	-	-
ho_noxs	1000	499500	4.33	0.02	0.22	0.64	8357741	0.00	0.55	0	989	8	974.42	0	-	-	-	-	-
hybrid	1000	315794	6.77	0.07	-	-	-	-	-	-	10	-	-	-	-	-	-	-	-
k	1000	499500	8.04	0.01	0.39	0.66	6261946	0.01	0.58	0	1353	-	-	-	-	-	-	-	-
ks	1000	499500	188.43	0.01	0.21	0.67	43506519	0.01	0.55	0	28624	-	-	-	-	-	318	-	-
ks_nopr	1000	499500	184.81	0.01	0.21	0.68	42122276	0.01	0.55	0	0	-	-	-	-	-	36067	-	-
ni	1000	499500	3.56	0.02	0.21	0.65	3154630	0.01	0.54	0	879	-	-	-	-	2	-	-	-
ni_nopr	1000	499500	4.92	0.09	0.21	0.65	6748303	0.10	-	-	-	-	-	-	-	10	-	-	-

Table A.12: NOI3 data

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	1000	24975	0.19	0.05	0.02	0.85	420812	0.06	0.04	0	41	5	673.81	1	950	-	-	-	-
ho_nopr	1000	24975	0.12	0.04	0.01	0.82	351518	0.08	-	-	-	7	632.03	1	990	-	-	-	-
ho_noprxs	1000	24975	0.40	0.05	0.01	0.89	1605808	0.06	-	-	-	119	314.38	879	-	-	-	-	-
ho_noxs	1000	24975	0.31	0.11	0.02	0.65	826219	0.11	0.03	0	748	45	434.81	204	-	-	-	-	-
hybrid	1000	24362	0.46	0.01	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	24975	0.43	0.03	0.02	0.61	419214	0.00	0.04	0	259	-	-	-	-	-	-	-	-
ks	1000	24975	6.14	0.04	0.02	0.60	1492234	0.04	0.04	0	10448	-	-	-	-	-	159	-	-
ks_nopr	1000	24975	6.11	0.10	0.01	0.69	1626503	0.07	0.04	0	0	-	-	-	-	-	1598	-	-
ni	1000	24975	0.14	0.03	0.02	0.45	138913	0.03	0.03	0	0	-	-	-	-	1	-	-	-
ni_nopr	1000	24975	0.11	0.06	0.02	0.42	138913	0.03	-	-	-	-	-	-	-	1	-	-	-
ho	1000	49950	0.46	0.03	0.24	0.72	1021161	0.03	0.07	0	115	4	690.95	0	877	-	-	-	-
ho_nopr	1000	49950	0.30	0.02	0.15	0.74	946597	0.02	-	-	-	10	549.76	0	988	-	-	-	-
ho_noprxs	1000	49950	0.70	0.06	0.15	0.71	2687387	0.06	-	-	-	118	379.10	881	-	-	-	-	-
ho_noxs	1000	49950	0.72	0.12	0.20	0.72	1885108	0.14	0.07	0	656	69	439.22	271	-	-	-	-	-
hybrid	1000	47529	0.75	0.01	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	49950	0.88	0.02	0.17	0.66	830775	0.01	0.08	0	557	-	-	-	-	-	-	-	-
ks	1000	49950	13.81	0.03	0.59	1.07	3336637	0.04	0.07	0	17308	-	-	-	-	-	159	-	-
ks_nopr	1000	49950	15.86	0.07	1.14	1.23	3933576	0.06	0.07	0	0	-	-	-	-	-	2980	-	-
ni	1000	49950	0.31	0.01	0.10	0.59	299629	0.02	0.07	0	1	-	-	-	-	2	-	-	-
ni_nopr	1000	49950	0.24	0.03	0.06	0.45	299633	0.02	-	-	-	-	-	-	-	2	-	-	-
ho	1000	124875	1.24	0.10	0.83	0.15	2663993	0.11	0.16	0	147	12	586.71	0	838	-	-	-	-
ho_nopr	1000	124875	0.90	0.04	0.60	0.13	2663778	0.03	-	-	-	16	535.05	0	982	-	-	-	-
ho_noprxs	1000	124875	1.44	0.03	0.57	0.14	5133037	0.04	-	-	-	193	379.41	805	-	-	-	-	-
ho_noxs	1000	124875	1.54	0.14	0.78	0.13	3656221	0.14	0.16	0	658	110	433.92	229	-	-	-	-	-
hybrid	1000	110503	1.56	0.03	-	-	-	-	-	-	4	-	-	-	-	-	-	-	-
k	1000	124875	2.19	0.01	0.67	0.03	1942652	0.00	0.18	0	736	-	-	-	-	-	-	-	-
ks	1000	124875	33.38	0.02	2.39	0.53	7846352	0.02	0.16	0	18120	-	-	-	-	-	162	-	-
ks_nopr	1000	124875	37.30	0.03	3.55	0.77	9021666	0.02	0.16	0	0	-	-	-	-	-	2988	-	-
ni	1000	124875	0.81	0.02	0.39	0.02	761927	0.03	0.16	0	87	-	-	-	-	2	-	-	-
ni_nopr	1000	124875	0.64	0.02	0.22	0.02	767477	0.03	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	2.58	0.10	1.65	0.17	5197277	0.12	0.30	0	177	11	681.56	0	810	-	-	-	-
ho_nopr	1000	249750	1.83	0.03	1.17	0.11	5145474	0.03	-	-	-	17	530.75	0	982	-	-	-	-
ho_noprxs	1000	249750	2.50	0.05	1.09	0.09	8270216	0.04	-	-	-	315	322.80	684	-	-	-	-	-
ho_noxs	1000	249750	2.91	0.03	1.56	0.12	6257279	0.02	0.31	0	752	128	403.56	117	-	-	-	-	-
hybrid	1000	196545	2.69	0.01	-	-	-	-	-	-	132	-	-	-	-	-	-	-	-
k	1000	249750	4.22	0.01	1.39	0.02	3485807	0.00	0.34	0	813	-	-	-	-	-	-	-	-
ks	1000	249750	59.42	0.03	4.40	0.71	13681153	0.03	0.31	0	17980	-	-	-	-	-	164	-	-
ks_nopr	1000	249750	63.73	0.02	4.47	0.60	15250038	0.02	0.31	0	0	-	-	-	-	-	2881	-	-
ni	1000	249750	1.54	0.01	0.77	0.01	1429950	0.01	0.30	0	156	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.24	0.02	0.47	0.01	1451158	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	1000	374625	3.67	0.01	2.36	0.25	7069828	0.03	0.44	0	254	10	675.26	0	733	-	-	-	-
ho_nopr	1000	374625	2.55	0.04	1.61	0.21	7192414	0.06	-	-	-	17	563.66	0	981	-	-	-	-
ho_noprxs	1000	374625	3.30	0.04	1.53	0.19	10473923	0.06	-	-	-	361	324.46	637	-	-	-	-	-
ho_noxs	1000	374625	3.79	0.06	2.26	0.23	7593893	0.06	0.44	0	880	90	481.60	27	-	-	-	-	-
hybrid	1000	263567	3.62	0.01	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	374625	6.04	0.01	2.13	0.02	4720956	0.00	0.48	0	849	-	-	-	-	-	-	-	-
ks	1000	374625	79.74	0.03	6.59	0.61	18162375	0.03	0.44	0	18043	-	-	-	-	-	164	-	-
ks_nopr	1000	374625	83.73	0.02	5.70	0.45	19909078	0.01	0.44	0	0	-	-	-	-	-	2879	-	-
ni	1000	374625	2.29	0.02	1.18	0.01	2001539	0.00	0.44	0	190	-	-	-	-	2	-	-	-
ni_nopr	1000	374625	1.85	0.02	0.74	0.01	2039467	0.00	-	-	-	-	-	-	-	2	-	-	-
ho	1000	499500	4.71	0.05	3.27	0.24	8659804	0.09	0.54	0	212	8	713.76	11	765	-	-	-	-
ho_nopr	1000	499500	3.19	0.05	2.10	0.19	8740015	0.09	-	-	-	16	578.23	11	970	-	-	-	-
ho_noprxs	1000	499500	4.10	0.05	2.04	0.17	12485081	0.06	-	-	-	408	314.83	590	-	-	-	-	-
ho_noxs	1000	499500	4.81	0.06	3.09	0.21	9246901	0.07	0.55	0	828	91	637.61	79	-	-	-	-	-
hybrid	1000	315794	4.35	0.01	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	499500	7.69	0.01	2.89	0.01	5694846	0.00	0.58	0	861	-	-	-	-	-	-	-	-
ks	1000	499500	95.12	0.04	7.63	0.96	21529012	0.04	0.55	0	17997	-	-	-	-	-	164	-	-
ks_nopr	1000	499500	100.02	0.03	8.31	0.86	23647290	0.03	0.55	0	0	-	-	-	-	-	3081	-	-
ni	1000	499500	2.91	0.01	1.55	0.01	2451567	0.01	0.54	0	195	-	-	-	-	2	-	-	-
ni_nopr	1000	499500	2.37	0.02	1.01	0.01	2497759	0.00	-	-	-	-	-	-	-	2	-	-	-

Table A.13: NOI4 data

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	1000	249750	2.10	0.02	0.16	0.62	4359927	0.01	0.31	0	754	4	997.52	0	238	-	-	-	-
ho_nopr	1000	249750	2.53	0.09	0.12	0.61	7605133	0.09	-	-	-	16	933.95	243	738	-	-	-	-
ho_noprxs	1000	249750	3.24	0.13	0.11	0.60	11050154	0.14	-	-	-	100	566.37	898	-	-	-	-	-
ho_noxs	1000	249750	2.10	0.02	0.16	0.61	4412605	0.02	0.31	0	964	7	854.90	26	-	-	-	-	-
hybrid	1000	196545	3.53	0.03	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-
k	1000	249750	4.41	0.01	0.25	0.56	3786958	0.00	0.34	0	1280	-	-	-	-	-	-	-	-
ks	1000	249750	121.28	0.01	0.15	0.59	28411005	0.01	0.31	0	25977	-	-	-	-	-	318	-	-
ks_nopr	1000	249750	114.81	0.01	0.15	0.60	26625662	0.01	0.31	0	0	-	-	-	-	-	26448	-	-
ni	1000	249750	1.88	0.02	0.15	0.60	1822543	0.01	0.30	0	787	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	2.16	0.04	0.16	0.60	2941612	0.05	-	-	-	-	-	-	-	7	-	-	-
ho	1000	249750	1.00	0.08	0.23	0.47	890688	0.25	0.63	998	0	1	2.00	0	0	-	-	-	-
ho_nopr	1000	249750	1.77	0.35	0.17	0.47	4705713	0.42	-	-	-	4	617.76	1	993	-	-	-	-
ho_noprxs	1000	249750	3.75	0.23	0.17	0.50	12853261	0.23	-	-	-	513	97.95	485	-	-	-	-	-
ho_noxs	1000	249750	0.99	0.09	0.23	0.46	890688	0.25	0.62	998	0	1	2.00	0	-	-	-	-	-
hybrid	1000	196545	2.46	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	249750	1.67	0.05	0.35	0.46	890687	0.25	1.13	998	0	-	-	-	-	-	-	-	-
ks	1000	249750	0.93	0.09	0.21	0.48	444391	0.51	0.58	998	-	-	-	-	-	-	0	-	-
ks_nopr	1000	249750	0.94	0.09	0.22	0.48	444391	0.51	0.59	998	-	-	-	-	-	-	0	-	-
ni	1000	249750	0.95	0.08	0.22	0.49	446295	0.00	0.60	998	0	-	-	-	-	0	-	-	-
ni_nopr	1000	249750	1.07	0.02	0.22	0.48	1050530	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	0.95	0.01	0.13	0.34	748225	0.01	0.58	998	0	1	2.00	0	0	-	-	-	-
ho_nopr	1000	249750	1.17	0.30	0.10	0.35	2841593	0.39	-	-	-	3	708.27	0	995	-	-	-	-
ho_noprxs	1000	249750	3.15	0.25	0.10	0.35	10623608	0.25	-	-	-	535	153.44	463	-	-	-	-	-
ho_noxs	1000	249750	0.94	0.01	0.14	0.36	748225	0.01	0.57	998	0	1	2.00	0	-	-	-	-	-
hybrid	1000	196545	2.43	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	249750	1.62	0.01	0.21	0.34	748224	0.01	1.07	998	0	-	-	-	-	-	-	-	-
ks	1000	249750	0.88	0.01	0.13	0.33	301929	0.02	0.54	998	-	-	-	-	-	-	0	-	-
ks_nopr	1000	249750	0.89	0.02	0.13	0.34	301929	0.02	0.54	998	-	-	-	-	-	-	0	-	-
ni	1000	249750	0.92	0.03	0.13	0.33	446295	0.00	0.55	998	0	-	-	-	-	0	-	-	-
ni_nopr	1000	249750	1.06	0.01	0.13	0.33	1045074	0.00	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	0.96	0.00	0.27	0.22	758572	0.00	0.59	998	0	1	2.00	0	0	-	-	-	-
ho_nopr	1000	249750	0.82	0.05	0.20	0.24	1725115	0.08	-	-	-	2	901.33	0	995	-	-	-	-
ho_noprxs	1000	249750	2.70	0.12	0.20	0.21	9060836	0.14	-	-	-	558	172.09	440	-	-	-	-	-
ho_noxs	1000	249750	0.94	0.01	0.27	0.22	758572	0.00	0.58	998	0	1	2.00	0	-	-	-	-	-
hybrid	1000	196545	2.48	0.01	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	249750	1.62	0.00	0.41	0.20	758571	0.00	1.08	998	0	-	-	-	-	-	-	-	-
ks	1000	249750	0.89	0.01	0.26	0.22	312275	0.01	0.54	998	-	-	-	-	-	-	0	-	-
ks_nopr	1000	249750	0.89	0.01	0.26	0.21	312275	0.01	0.54	998	-	-	-	-	-	-	0	-	-
ni	1000	249750	0.91	0.01	0.26	0.22	446295	0.00	0.56	998	0	-	-	-	-	0	-	-	-
ni_nopr	1000	249750	1.06	0.01	0.26	0.22	1045504	0.00	-	-	-	-	-	-	-	1	-	-	-
ho	1000	249750	1.01	0.01	0.17	0.46	872233	0.00	0.64	998	0	1	2.00	0	0	-	-	-	-
ho_nopr	1000	249750	0.90	0.06	0.13	0.56	1902627	0.05	-	-	-	2	968.32	0	995	-	-	-	-
ho_noprxs	1000	249750	4.43	0.54	0.12	0.46	15203098	0.54	-	-	-	615	90.04	383	-	-	-	-	-
ho_noxs	1000	249750	0.99	0.02	0.16	0.45	872233	0.00	0.63	998	0	1	2.00	0	-	-	-	-	-
hybrid	1000	196545	2.61	0.01	-	-	-	-	-	-	6	-	-	-	-	-	-	-	-
k	1000	249750	1.66	0.01	0.26	0.43	872232	0.00	1.11	998	0	-	-	-	-	-	-	-	-
ks	1000	249750	0.94	0.01	0.16	0.47	425936	0.01	0.58	998	-	-	-	-	-	-	0	-	-
ks_nopr	1000	249750	0.93	0.02	0.16	0.46	425936	0.01	0.59	998	-	-	-	-	-	-	0	-	-
ni	1000	249750	0.96	0.01	0.16	0.45	446295	0.00	0.60	998	0	-	-	-	-	0	-	-	-
ni_nopr	1000	249750	1.04	0.00	0.16	0.45	1055629	0.00	-	-	-	-	-	-	-	1	-	-	-
ho	1000	249750	1.09	0.01	0.19	0.50	1131721	0.01	0.72	998	0	1	2.00	0	0	-	-	-	-
ho_nopr	1000	249750	1.10	0.07	0.14	0.50	2387355	0.05	-	-	-	3	920.46	11	984	-	-	-	-
ho_noprxs	1000	249750	4.32	0.08	0.13	0.49	14553432	0.07	-	-	-	606	85.72	392	-	-	-	-	-
ho_noxs	1000	249750	1.06	0.01	0.18	0.51	1131721	0.01	0.70	998	0	1	2.00	0	-	-	-	-	-
hybrid	1000	196545	2.76	0.01	-	-	-	-	-	-	74	-	-	-	-	-	-	-	-
k	1000	249750	1.73	0.01	0.29	0.48	1131720	0.01	1.19	998	0	-	-	-	-	-	-	-	-
ks	1000	249750	1.01	0.01	0.17	0.51	685425	0.02	0.67	998	-	-	-	-	-	-	0	-	-
ks_nopr	1000	249750	1.02	0.01	0.17	0.50	685425	0.02	0.67	998	-	-	-	-	-	-	0	-	-
ni	1000	249750	1.03	0.01	0.17	0.51	446295	0.00	0.68	998	0	-	-	-	-	0	-	-	-
ni_nopr	1000	249750	1.09	0.03	0.18	0.47	1157227	0.02	-	-	-	-	-	-	-	1	-	-	-

Table A.14: NOI5 data

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	1000	249750	1.06	0.01	0.22	0.17	1128305	0.01	0.69	998	0	1	2.00	0	0	-	-	-	-
ho_nopr	1000	249750	1.22	0.07	0.16	0.13	2839732	0.04	-	-	-	4	965.36	38	956	-	-	-	-
ho_noprxs	1000	249750	5.29	0.16	0.16	0.15	18173266	0.18	-	-	-	546	83.87	452	-	-	-	-	-
ho_noxs	1000	249750	1.04	0.00	0.22	0.14	1128305	0.01	0.68	998	0	1	2.00	0	-	-	-	-	-
hybrid	1000	196545	2.96	0.03	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	249750	1.72	0.01	0.34	0.13	1128304	0.01	1.17	998	0	-	-	-	-	-	-	-	-
ks	1000	249750	0.99	0.01	0.21	0.14	682009	0.01	0.64	998	-	-	-	-	-	-	0	-	-
ks_nopr	1000	249750	1.00	0.02	0.21	0.15	682009	0.01	0.65	998	-	-	-	-	-	-	0	-	-
ni	1000	249750	1.01	0.00	0.21	0.13	446295	0.00	0.65	998	0	-	-	-	-	0	-	-	-
ni_nopr	1000	249750	1.16	0.03	0.21	0.16	1305785	0.02	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	1.08	0.02	0.26	0.17	1183977	0.01	0.71	998	0	1	2.00	0	0	-	-	-	-
ho_nopr	1000	249750	1.52	0.03	0.18	0.13	3796370	0.04	-	-	-	6	965.51	50	942	-	-	-	-
ho_noprxs	1000	249750	4.23	0.24	0.19	0.18	14478060	0.23	-	-	-	495	113.90	503	-	-	-	-	-
ho_noxs	1000	249750	1.05	0.03	0.26	0.16	1183977	0.01	0.69	998	0	1	2.00	0	-	-	-	-	-
hybrid	1000	196545	2.96	0.01	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-
k	1000	249750	1.73	0.01	0.40	0.15	1183976	0.01	1.18	998	0	-	-	-	-	-	-	-	-
ks	1000	249750	1.00	0.01	0.24	0.15	737680	0.01	0.65	998	-	-	-	-	-	-	0	-	-
ks_nopr	1000	249750	1.00	0.02	0.24	0.16	737680	0.01	0.65	998	-	-	-	-	-	-	0	-	-
ni	1000	249750	1.02	0.01	0.25	0.15	446295	0.00	0.66	998	0	-	-	-	-	0	-	-	-
ni_nopr	1000	249750	1.26	0.03	0.25	0.17	1395155	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	1.07	0.01	0.19	0.59	1215232	0.00	0.70	998	0	1	2.00	0	0	-	-	-	-
ho_nopr	1000	249750	1.69	0.13	0.14	0.56	4437649	0.16	-	-	-	6	909.58	34	957	-	-	-	-
ho_noprxs	1000	249750	4.72	0.11	0.14	0.60	16244958	0.12	-	-	-	442	115.53	556	-	-	-	-	-
ho_noxs	1000	249750	1.06	0.00	0.19	0.58	1215232	0.00	0.69	998	0	1	2.00	0	-	-	-	-	-
hybrid	1000	196545	3.05	0.03	-	-	-	-	-	-	10	-	-	-	-	-	-	-	-
k	1000	249750	1.73	0.01	0.29	0.57	1215231	0.00	1.19	998	0	-	-	-	-	-	-	-	-
ks	1000	249750	1.00	0.01	0.18	0.58	768936	0.01	0.65	998	-	-	-	-	-	-	0	-	-
ks_nopr	1000	249750	1.00	0.01	0.18	0.58	768936	0.01	0.66	998	-	-	-	-	-	-	0	-	-
ni	1000	249750	1.01	0.01	0.18	0.59	446295	0.00	0.66	998	0	-	-	-	-	0	-	-	-
ni_nopr	1000	249750	1.38	0.02	0.19	0.57	1566652	0.07	-	-	-	-	-	-	-	3	-	-	-
ho	1000	249750	2.59	0.10	1.66	0.17	5197277	0.12	0.30	0	177	11	681.56	0	810	-	-	-	-
ho_nopr	1000	249750	1.81	0.02	1.15	0.13	5145474	0.03	-	-	-	17	530.75	0	982	-	-	-	-
ho_noprxs	1000	249750	2.53	0.04	1.10	0.07	8270216	0.04	-	-	-	315	322.80	684	-	-	-	-	-
ho_noxs	1000	249750	2.92	0.04	1.56	0.11	6257279	0.02	0.31	0	752	128	403.56	117	-	-	-	-	-
hybrid	1000	196545	2.68	0.01	-	-	-	-	-	-	132	-	-	-	-	-	-	-	-
k	1000	249750	4.21	0.01	1.38	0.02	3485807	0.00	0.34	0	813	-	-	-	-	-	-	-	-
ks	1000	249750	59.45	0.03	4.41	0.71	13681153	0.03	0.31	0	17980	-	-	-	-	-	164	-	-
ks_nopr	1000	249750	63.78	0.02	4.47	0.60	15250038	0.02	0.31	0	0	-	-	-	-	-	2881	-	-
ni	1000	249750	1.54	0.01	0.77	0.01	1429950	0.01	0.30	0	156	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.25	0.01	0.47	0.02	1451158	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	2.50	0.16	1.49	0.52	4960444	0.19	0.31	0	129	24	374.13	2	843	-	-	-	-
ho_nopr	1000	249750	1.95	0.06	1.10	0.50	5842424	0.06	-	-	-	25	391.76	7	966	-	-	-	-
ho_noprxs	1000	249750	2.97	0.04	1.20	0.58	10187248	0.04	-	-	-	220	230.46	778	-	-	-	-	-
ho_noxs	1000	249750	2.78	0.16	1.48	0.53	6011778	0.19	0.31	0	678	96	274.36	223	-	-	-	-	-
hybrid	1000	196545	2.63	0.03	-	-	-	-	-	-	18	-	-	-	-	-	-	-	-
k	1000	249750	19.25	0.12	1.79	0.53	13475576	0.09	0.34	0	478	-	-	-	-	-	-	2.27	12.71
ks	1000	249750	74.78	0.07	3.73	0.89	18332202	0.07	0.31	0	22360	-	-	-	-	-	196	-	-
ks_nopr	1000	249750	83.11	0.04	10.16	0.96	19218594	0.04	0.31	0	0	-	-	-	-	-	7266	-	-
ni	1000	249750	1.63	0.04	0.97	0.52	1440661	0.03	0.31	0	302	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.31	0.04	0.71	0.54	1512715	0.05	-	-	-	-	-	-	-	4	-	-	-
ho	1000	249750	2.89	0.03	0.25	0.36	6036279	0.04	0.31	0	90	29	265.54	6	872	-	-	-	-
ho_nopr	1000	249750	2.36	0.03	0.19	0.38	7179069	0.05	-	-	-	33	275.50	11	954	-	-	-	-
ho_noprxs	1000	249750	3.17	0.03	0.18	0.40	11256572	0.03	-	-	-	249	147.41	749	-	-	-	-	-
ho_noxs	1000	249750	3.58	0.08	0.25	0.36	8264702	0.08	0.31	0	563	152	181.82	282	-	-	-	-	-
hybrid	1000	196545	2.59	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	249750	17.29	0.61	0.38	0.34	11854675	0.55	0.34	0	542	-	-	-	-	-	-	1.29	10.90
ks	1000	249750	69.33	0.03	0.24	0.36	17094353	0.04	0.31	0	23390	-	-	-	-	-	204	-	-
ks_nopr	1000	249750	80.88	0.03	0.24	0.38	18617982	0.03	0.31	0	0	-	-	-	-	-	6554	-	-
ni	1000	249750	1.55	0.02	0.25	0.37	1374397	0.01	0.31	0	198	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.26	0.01	0.25	0.35	1406916	0.02	-	-	-	-	-	-	-	5	-	-	-

Table A.15: NOI5 data (cont)

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	1000	249750	3.18	0.06	0.18	0.45	6718974	0.07	0.31	0	104	25	292.36	2	867	-	-	-	-
ho_nopr	1000	249750	2.62	0.07	0.13	0.46	8036162	0.09	-	-	-	32	262.58	7	959	-	-	-	-
ho_noprxs	1000	249750	3.60	0.02	0.13	0.48	12805103	0.02	-	-	-	279	134.78	719	-	-	-	-	-
ho_noxs	1000	249750	4.03	0.07	0.18	0.45	9413915	0.08	0.31	0	587	172	135.96	238	-	-	-	-	-
hybrid	1000	196545	2.56	0.02	-	-	-	-	-	-	9	-	-	-	-	-	-	-	-
k	1000	249750	4.45	0.03	0.28	0.43	3832165	0.08	0.34	0	765	-	-	-	-	-	-	-	-
ks	1000	249750	66.06	0.04	0.17	0.47	15842343	0.06	0.31	0	22132	-	-	-	-	-	204	-	-
ks_nopr	1000	249750	76.88	0.06	0.17	0.46	17686963	0.05	0.31	0	0	-	-	-	-	-	5669	-	-
ni	1000	249750	1.53	0.02	0.17	0.46	1328932	0.03	0.31	0	115	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.23	0.02	0.17	0.44	1342940	0.04	-	-	-	-	-	-	-	5	-	-	-
ho	1000	249750	3.04	0.14	0.22	0.43	6114897	0.21	0.35	47	43	11	330.10	1	895	-	-	-	-
ho_nopr	1000	249750	2.20	0.28	0.16	0.45	6511378	0.31	-	-	-	19	294.41	0	978	-	-	-	-
ho_noprxs	1000	249750	3.40	0.11	0.15	0.45	11981642	0.11	-	-	-	346	95.20	652	-	-	-	-	-
ho_noxs	1000	249750	3.62	0.08	0.21	0.42	7819953	0.10	0.35	47	659	158	116.97	133	-	-	-	-	-
hybrid	1000	196545	2.43	0.02	-	-	-	-	-	-	6	-	-	-	-	-	-	-	-
k	1000	249750	4.14	0.03	0.34	0.40	3509270	0.11	0.62	58	463	-	-	-	-	-	-	-	-
ks	1000	249750	51.58	0.10	0.21	0.44	11885487	0.08	0.52	58	14907	-	-	-	-	-	167	-	-
ks_nopr	1000	249750	54.61	0.14	0.21	0.43	13243441	0.10	0.52	58	0	-	-	-	-	-	2535	-	-
ni	1000	249750	1.44	0.01	0.22	0.46	1112212	0.05	0.35	47	8	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.14	0.02	0.21	0.43	1184449	0.03	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	2.49	0.44	0.17	0.79	4658657	0.61	0.41	235	24	4	500.42	0	733	-	-	-	-
ho_nopr	1000	249750	2.03	0.32	0.12	0.81	5656390	0.37	-	-	-	6	469.87	1	991	-	-	-	-
ho_noprxs	1000	249750	3.63	0.19	0.12	0.81	12421927	0.19	-	-	-	428	102.21	570	-	-	-	-	-
ho_noxs	1000	249750	3.19	0.45	0.17	0.80	6742579	0.56	0.41	235	641	62	224.32	58	-	-	-	-	-
hybrid	1000	196545	2.43	0.02	-	-	-	-	-	-	3	-	-	-	-	-	-	-	-
k	1000	249750	3.52	0.27	0.26	0.76	2602874	0.36	0.56	235	147	-	-	-	-	-	-	-	-
ks	1000	249750	37.81	0.50	0.16	0.80	8651244	0.49	0.39	235	6207	-	-	-	-	-	127	-	-
ks_nopr	1000	249750	35.49	0.51	0.16	0.80	8998740	0.49	0.40	235	0	-	-	-	-	-	998	-	-
ni	1000	249750	1.29	0.16	0.17	0.80	908743	0.26	0.40	235	3	-	-	-	-	1	-	-	-
ni_nopr	1000	249750	1.08	0.01	0.16	0.77	1072016	0.02	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	2.10	0.29	0.24	0.49	3583047	0.44	0.43	276	7	4	436.91	0	710	-	-	-	-
ho_nopr	1000	249750	2.14	0.15	0.17	0.49	6119261	0.19	-	-	-	6	587.72	0	992	-	-	-	-
ho_noprxs	1000	249750	4.08	0.19	0.18	0.49	14013247	0.18	-	-	-	490	78.77	508	-	-	-	-	-
ho_noxs	1000	249750	3.32	0.41	0.24	0.47	6987824	0.50	0.43	276	638	39	227.87	44	-	-	-	-	-
hybrid	1000	196545	2.45	0.01	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-
k	1000	249750	3.58	0.28	0.36	0.48	3157705	0.38	1.13	335	84	-	-	-	-	-	-	-	-
ks	1000	249750	28.49	0.53	0.22	0.48	7578667	0.51	0.85	335	4209	-	-	-	-	-	124	-	-
ks_nopr	1000	249750	25.99	0.53	0.23	0.48	7711374	0.51	0.85	335	0	-	-	-	-	-	742	-	-
ni	1000	249750	1.26	0.14	0.23	0.49	860232	0.24	0.42	276	3	-	-	-	-	1	-	-	-
ni_nopr	1000	249750	1.08	0.01	0.23	0.47	1057393	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	1.35	0.55	0.14	0.48	1760189	1.07	0.56	810	11	1	78.56	0	175	-	-	-	-
ho_nopr	1000	249750	2.29	0.48	0.10	0.50	6423214	0.55	-	-	-	5	555.73	1	991	-	-	-	-
ho_noprxs	1000	249750	4.24	0.06	0.09	0.54	14737105	0.07	-	-	-	508	55.50	490	-	-	-	-	-
ho_noxs	1000	249750	1.70	0.88	0.14	0.49	2872487	1.43	0.55	810	127	31	22.88	30	-	-	-	-	-
hybrid	1000	196545	2.45	0.01	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	1000	249750	2.16	0.47	0.22	0.49	1386311	0.81	1.05	816	33	-	-	-	-	-	-	-	-
ks	1000	249750	9.84	1.82	0.13	0.50	2441019	1.69	0.59	816	1806	-	-	-	-	-	31	-	-
ks_nopr	1000	249750	9.46	1.81	0.13	0.51	2543921	1.70	0.59	816	0	-	-	-	-	-	336	-	-
ni	1000	249750	1.05	0.18	0.14	0.51	561173	0.41	0.54	810	1	-	-	-	-	0	-	-	-
ni_nopr	1000	249750	1.08	0.01	0.13	0.48	1054870	0.01	-	-	-	-	-	-	-	2	-	-	-

Table A.16: NOI5 data (cont)

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	1000	249750	2.26	0.09	1.80	0.15	4339266	0.09	0.31	0	25	1	999.53	0	970	-	-	-	-
ho_nopr	1000	249750	1.34	0.08	1.20	0.09	3614985	0.08	-	-	-	5	682.73	0	994	-	-	-	-
ho_noprxs	1000	249750	1.84	0.05	1.00	0.08	5714081	0.06	-	-	-	423	320.91	575	-	-	-	-	-
ho_noxs	1000	249750	2.09	0.08	1.50	0.11	4247732	0.10	0.31	0	992	3	638.70	2	-	-	-	-	-
hybrid	1000	196545	2.37	0.01	-	-	-	-	-	-	393	-	-	-	-	-	-	-	-
k	1000	249750	4.07	0.01	1.39	0.02	3388096	0.00	0.34	0	250	-	-	-	-	-	-	-	-
ks	1000	249750	44.74	0.01	2.70	0.16	11215213	0.01	0.30	0	276	-	-	-	-	-	136	-	-
ks_nopr	1000	249750	38.80	0.02	2.42	0.21	11143615	0.01	0.31	0	0	-	-	-	-	-	209	-	-
ni	1000	249750	1.44	0.02	0.78	0.01	1262330	0.01	0.31	0	0	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.12	0.01	0.47	0.02	1262330	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	2.16	0.07	1.62	0.12	4192595	0.08	0.31	0	117	2	999.27	0	879	-	-	-	-
ho_nopr	1000	249750	1.47	0.02	1.13	0.06	4034558	0.06	-	-	-	11	567.39	0	987	-	-	-	-
ho_noprxs	1000	249750	2.13	0.03	1.07	0.03	6815571	0.04	-	-	-	352	345.03	646	-	-	-	-	-
ho_noxs	1000	249750	2.12	0.07	1.54	0.10	4301366	0.10	0.31	0	992	3	667.56	3	-	-	-	-	-
hybrid	1000	196545	2.66	0.02	-	-	-	-	-	-	11	-	-	-	-	-	-	-	-
k	1000	249750	4.13	0.01	1.40	0.02	3394484	0.00	0.34	0	476	-	-	-	-	-	-	-	-
ks	1000	249750	48.27	0.02	3.44	0.42	11634480	0.02	0.30	0	4189	-	-	-	-	-	163	-	-
ks_nopr	1000	249750	42.59	0.02	3.10	0.29	11639741	0.01	0.30	0	0	-	-	-	-	-	649	-	-
ni	1000	249750	1.47	0.02	0.77	0.01	1318399	0.01	0.31	0	15	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.17	0.02	0.47	0.02	1318646	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	2.59	0.09	1.66	0.18	5197277	0.12	0.31	0	177	11	681.56	0	810	-	-	-	-
ho_nopr	1000	249750	1.82	0.01	1.16	0.12	5145474	0.03	-	-	-	17	530.75	0	982	-	-	-	-
ho_noprxs	1000	249750	2.52	0.03	1.10	0.07	8270216	0.04	-	-	-	315	322.80	684	-	-	-	-	-
ho_noxs	1000	249750	2.91	0.04	1.56	0.11	6257279	0.02	0.31	0	752	128	403.56	117	-	-	-	-	-
hybrid	1000	196545	2.69	0.01	-	-	-	-	-	-	132	-	-	-	-	-	-	-	-
k	1000	249750	4.23	0.01	1.40	0.02	3485807	0.00	0.34	0	813	-	-	-	-	-	-	-	-
ks	1000	249750	59.34	0.03	4.39	0.71	13681153	0.03	0.31	0	17980	-	-	-	-	-	164	-	-
ks_nopr	1000	249750	63.72	0.02	4.46	0.60	15250038	0.02	0.31	0	0	-	-	-	-	-	2881	-	-
ni	1000	249750	1.56	0.03	0.79	0.02	1429950	0.01	0.31	0	156	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.24	0.01	0.47	0.01	1451158	0.01	-	-	-	-	-	-	-	2	-	-	-
ho	1000	249750	2.88	0.07	0.19	0.53	6064750	0.11	0.31	0	258	13	633.26	40	686	-	-	-	-
ho_nopr	1000	249750	2.29	0.11	0.14	0.56	6996101	0.16	-	-	-	20	576.13	88	890	-	-	-	-
ho_noprxs	1000	249750	3.28	0.05	0.14	0.54	11617066	0.06	-	-	-	155	297.74	844	-	-	-	-	-
ho_noxs	1000	249750	3.11	0.07	0.19	0.52	7010881	0.09	0.31	0	734	54	416.23	208	-	-	-	-	-
hybrid	1000	196545	3.07	0.06	-	-	-	-	-	-	59	-	-	-	-	-	-	-	-
k	1000	249750	4.44	0.01	0.29	0.50	3806190	0.02	0.34	0	1172	-	-	-	-	-	-	-	-
ks	1000	249750	113.91	0.02	0.18	0.54	26299972	0.03	0.31	0	22657	-	-	-	-	-	317	-	-
ks_nopr	1000	249750	104.73	0.03	0.18	0.53	24202592	0.03	0.31	0	0	-	-	-	-	-	17932	-	-
ni	1000	249750	1.81	0.03	0.18	0.55	1700141	0.03	0.31	0	653	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.72	0.08	0.18	0.51	2212316	0.11	-	-	-	-	-	-	-	5	-	-	-
ho	1000	249750	2.55	0.05	0.18	0.48	5137483	0.04	0.31	0	474	5	818.03	39	480	-	-	-	-
ho_nopr	1000	249750	2.21	0.08	0.13	0.45	6576748	0.08	-	-	-	17	617.23	78	903	-	-	-	-
ho_noprxs	1000	249750	3.28	0.06	0.13	0.46	11351682	0.06	-	-	-	123	338.70	876	-	-	-	-	-
ho_noxs	1000	249750	2.99	0.17	0.18	0.46	6629090	0.18	0.31	0	786	36	535.06	175	-	-	-	-	-
hybrid	1000	196545	3.11	0.06	-	-	-	-	-	-	76	-	-	-	-	-	-	-	-
k	1000	249750	4.43	0.01	0.28	0.44	3778360	0.01	0.34	0	1184	-	-	-	-	-	-	-	-
ks	1000	249750	114.18	0.03	0.17	0.46	26504913	0.03	0.31	0	23041	-	-	-	-	-	317	-	-
ks_nopr	1000	249750	105.56	0.03	0.17	0.45	24422932	0.03	0.31	0	0	-	-	-	-	-	18871	-	-
ni	1000	249750	1.82	0.04	0.17	0.45	1715849	0.03	0.31	0	670	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.75	0.08	0.17	0.45	2282932	0.10	-	-	-	-	-	-	-	5	-	-	-
ho	1000	249750	2.27	0.08	0.20	0.61	4521039	0.07	0.31	0	637	5	971.33	0	355	-	-	-	-
ho_nopr	1000	249750	2.10	0.06	0.14	0.61	6069733	0.05	-	-	-	18	685.29	68	912	-	-	-	-
ho_noprxs	1000	249750	3.17	0.06	0.14	0.65	10867025	0.08	-	-	-	136	380.92	862	-	-	-	-	-
ho_noxs	1000	249750	2.43	0.24	0.19	0.62	5163926	0.29	0.31	0	847	18	874.05	132	-	-	-	-	-
hybrid	1000	196545	2.93	0.03	-	-	-	-	-	-	3	-	-	-	-	-	-	-	-
k	1000	249750	4.38	0.01	0.29	0.59	3657228	0.01	0.34	0	1129	-	-	-	-	-	-	-	-
ks	1000	249750	109.85	0.03	0.18	0.61	25502234	0.03	0.31	0	21547	-	-	-	-	-	317	-	-
ks_nopr	1000	249750	100.18	0.03	0.18	0.61	23377910	0.03	0.31	0	0	-	-	-	-	-	15414	-	-
ni	1000	249750	1.76	0.02	0.18	0.61	1649211	0.02	0.31	0	593	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.63	0.04	0.19	0.59	2009053	0.06	-	-	-	-	-	-	-	4	-	-	-

Table A.17: NOI6 data

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	1000	249750	2.15	0.02	0.17	0.59	4376733	0.02	0.30	0	648	4	996.65	0	344	-	-	-	-
ho_nopr	1000	249750	1.99	0.04	0.12	0.63	5768994	0.05	-	-	-	15	839.42	61	922	-	-	-	-
ho_noprxs	1000	249750	3.25	0.05	0.12	0.63	11248585	0.06	-	-	-	116	420.56	883	-	-	-	-	-
ho_noxs	1000	249750	2.41	0.25	0.16	0.61	5201591	0.31	0.31	0	906	10	877.08	81	-	-	-	-	-
hybrid	1000	196545	3.04	0.03	-	-	-	-	-	-	3	-	-	-	-	-	-	-	-
k	1000	249750	4.38	0.01	0.26	0.56	3685733	0.01	0.34	0	1158	-	-	-	-	-	-	-	-
ks	1000	249750	112.25	0.02	0.15	0.61	26083341	0.02	0.30	0	22293	-	-	-	-	-	318	-	-
ks_nopr	1000	249750	102.33	0.03	0.16	0.61	23896519	0.02	0.31	0	0	-	-	-	-	-	16968	-	-
ni	1000	249750	1.78	0.02	0.16	0.61	1688875	0.02	0.31	0	639	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	1.72	0.05	0.16	0.61	2144898	0.06	-	-	-	-	-	-	-	5	-	-	-
ho	1000	249750	2.13	0.04	0.24	0.40	4484626	0.06	0.30	0	673	5	997.71	0	320	-	-	-	-
ho_nopr	1000	249750	2.24	0.05	0.17	0.41	6684792	0.07	-	-	-	13	925.08	118	867	-	-	-	-
ho_noprxs	1000	249750	3.28	0.02	0.17	0.40	11220557	0.04	-	-	-	111	531.62	887	-	-	-	-	-
ho_noxs	1000	249750	2.20	0.11	0.24	0.38	4753967	0.12	0.31	0	889	6	842.02	102	-	-	-	-	-
hybrid	1000	196545	3.51	0.08	-	-	-	-	-	-	5	-	-	-	-	-	-	-	-
k	1000	249750	4.43	0.01	0.37	0.37	3748140	0.00	0.34	0	1239	-	-	-	-	-	-	-	-
ks	1000	249750	115.21	0.02	0.23	0.40	26840123	0.02	0.31	0	23305	-	-	-	-	-	318	-	-
ks_nopr	1000	249750	105.84	0.03	0.23	0.38	24667815	0.03	0.31	0	0	-	-	-	-	-	19321	-	-
ni	1000	249750	1.91	0.02	0.24	0.38	1830892	0.02	0.31	0	798	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	2.44	0.16	0.28	0.49	2896334	0.14	-	-	-	-	-	-	-	6	-	-	-
ho	1000	249750	2.10	0.01	0.21	0.27	4330353	0.01	0.31	0	753	4	948.17	0	240	-	-	-	-
ho_nopr	1000	249750	2.38	0.04	0.14	0.26	7337585	0.04	-	-	-	19	919.38	256	724	-	-	-	-
ho_noprxs	1000	249750	3.22	0.09	0.14	0.29	11153276	0.09	-	-	-	108	596.73	890	-	-	-	-	-
ho_noxs	1000	249750	2.07	0.01	0.20	0.26	4352467	0.01	0.31	0	943	6	837.24	49	-	-	-	-	-
hybrid	1000	196545	3.45	0.05	-	-	-	-	-	-	7	-	-	-	-	-	-	-	-
k	1000	249750	4.39	0.01	0.32	0.25	3774942	0.01	0.34	0	1267	-	-	-	-	-	-	-	-
ks	1000	249750	120.74	0.01	0.19	0.26	28263455	0.01	0.31	0	25701	-	-	-	-	-	318	-	-
ks_nopr	1000	249750	113.70	0.02	0.19	0.27	26368168	0.02	0.30	0	0	-	-	-	-	-	25369	-	-
ni	1000	249750	1.88	0.02	0.20	0.27	1802856	0.02	0.30	0	768	-	-	-	-	2	-	-	-
ni_nopr	1000	249750	2.06	0.08	0.19	0.26	2826420	0.09	-	-	-	-	-	-	-	6	-	-	-

Table A.18: NOI6 data (cont)

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	1000	4000	0.10	0.15	0.00	0.00	301923	0.15	0.01	0	566	13	848.65	69	349	-	-	-	-
ho_nopr	1000	4000	0.10	0.23	0.00	0.00	368658	0.24	-	-	-	24	614.17	383	591	-	-	-	-
ho_noprxs	1000	4000	0.11	0.14	0.00	0.00	438589	0.14	-	-	-	108	214.51	890	-	-	-	-	-
ho_noxs	1000	4000	0.12	0.08	0.00	0.00	436247	0.10	0.01	0	679	37	537.12	281	-	-	-	-	-
hybrid	1000	3990	3.75	0.02	-	-	-	-	-	-	14	-	-	-	-	-	-	-	-
k	1000	4000	0.15	0.17	0.00	0.00	127508	0.14	0.01	0	143	-	-	-	-	-	-	-	-
ks	1000	4000	3.55	0.02	0.00	0.00	1264952	0.01	0.01	0	35495	-	-	-	-	-	395	-	-
ks_nopr	1000	4000	6.93	0.02	0.00	0.00	1721490	0.01	0.01	0	0	-	-	-	-	-	77910	-	-
ni	1000	4000	3.40	0.02	0.00	0.00	4513288	0.02	0.00	0	476	-	-	-	-	297	-	-	-
ni_nopr	1000	4000	2.80	0.04	0.00	0.00	4582849	0.02	-	-	-	-	-	-	-	324	-	-	-
ho	2000	64000	1.80	0.06	0.00	0.00	5168993	0.06	0.09	0	1577	14	1534.99	295	111	-	-	-	-
ho_nopr	2000	64000	1.59	0.06	0.00	2.00	6088960	0.06	-	-	-	18	1465.16	1744	237	-	-	-	-
ho_noprxs	2000	64000	1.36	0.04	0.00	0.00	5564455	0.03	-	-	-	39	787.24	1959	-	-	-	-	-
ho_noxs	2000	64000	1.70	0.05	0.00	0.00	5022326	0.06	0.09	0	1660	25	1024.36	312	-	-	-	-	-
hybrid	2000	62984	114.57	0.01	-	-	-	-	-	-	316	-	-	-	-	-	-	-	-
k	2000	64000	4.84	0.07	0.00	4.90	1563491	0.13	0.11	0	903	-	-	-	-	-	-	-	-
ks	2000	64000	62.46	0.00	0.00	0.00	15558856	0.00	0.09	0	73880	-	-	-	-	-	336	-	-
ks_nopr	2000	64000	70.15	0.01	0.00	0.00	16469977	0.00	0.09	0	0	-	-	-	-	-	161924	-	-
ni	2000	64000	116.79	0.01	0.00	0.00	152144327	0.01	0.09	0	1362	-	-	-	-	615	-	-	-
ni_nopr	2000	64000	91.25	0.01	0.00	0.00	158207301	0.01	-	-	-	-	-	-	-	943	-	-	-
ho	2000	128000	3.61	0.07	0.00	0.00	10111352	0.07	0.18	0	1517	12	1782.44	390	77	-	-	-	-
ho_nopr	2000	128000	3.08	0.03	0.00	0.00	12220563	0.06	-	-	-	17	1529.17	1859	121	-	-	-	-
ho_noprxs	2000	128000	2.66	0.11	0.00	0.00	10423576	0.10	-	-	-	30	924.68	1968	-	-	-	-	-
ho_noxs	2000	128000	3.18	0.15	0.00	0.00	8951626	0.17	0.18	0	1833	18	1202.97	145	-	-	-	-	-
hybrid	2000	124048	209.61	0.01	-	-	-	-	-	-	370	-	-	-	-	-	-	-	-
k	2000	128000	13.89	0.02	0.00	0.00	2826500	0.01	0.20	0	915	-	-	-	-	-	-	-	-
ks	2000	128000	117.79	0.00	0.00	0.00	29037211	0.00	0.18	0	73483	-	-	-	-	-	336	-	-
ks_nopr	2000	128000	126.64	0.01	0.00	0.00	29606137	0.01	0.18	0	0	-	-	-	-	-	161430	-	-
ni	2000	128000	218.45	0.01	0.00	2.00	285855580	0.01	0.17	0	1393	-	-	-	-	590	-	-	-
ni_nopr	2000	128000	172.07	0.01	0.00	0.00	299663100	0.00	-	-	-	-	-	-	-	973	-	-	-
ho	2000	256000	6.41	0.08	0.00	2.00	16640099	0.10	0.35	0	1384	11	1605.93	558	44	-	-	-	-
ho_nopr	2000	256000	6.07	0.09	0.00	1.22	23310150	0.09	-	-	-	14	1682.14	1910	74	-	-	-	-
ho_noprxs	2000	256000	6.11	0.12	0.00	0.00	23142345	0.11	-	-	-	26	983.57	1972	-	-	-	-	-
ho_noxs	2000	256000	5.78	0.15	0.00	2.00	15858456	0.17	0.35	0	1796	15	1214.57	186	-	-	-	-	-
hybrid	2000	240400	379.74	0.01	-	-	-	-	-	-	407	-	-	-	-	-	-	-	-
k	2000	256000	93.16	0.02	0.00	2.71	69687532	0.02	0.40	0	935	-	-	-	-	-	-	5.69	82.80
ks	2000	256000	213.74	0.01	0.00	0.00	52257106	0.01	0.35	0	73218	-	-	-	-	-	336	-	-
ks_nopr	2000	256000	226.50	0.01	0.00	4.90	52735033	0.00	0.35	0	0	-	-	-	-	-	161456	-	-
ni	2000	256000	417.78	0.02	0.00	0.00	524331685	0.00	0.35	0	1420	-	-	-	-	568	-	-	-
ni_nopr	2000	256000	324.09	0.01	0.00	0.00	555036368	0.00	-	-	-	-	-	-	-	988	-	-	-
ho	4000	16000	0.96	0.36	0.00	0.00	1904097	0.37	0.04	0	2210	21	3453.75	469	1296	-	-	-	-
ho_nopr	4000	16000	0.60	0.22	0.00	2.00	1520746	0.23	-	-	-	26	2569.89	1756	2216	-	-	-	-
ho_noprxs	4000	16000	0.82	0.08	0.00	0.00	2178736	0.09	-	-	-	288	394.62	3711	-	-	-	-	-
ho_noxs	4000	16000	1.06	0.13	0.00	0.00	2179954	0.13	0.04	0	2840	52	1862.11	1104	-	-	-	-	-
hybrid	4000	15988	79.00	0.02	-	-	-	-	-	-	40	-	-	-	-	-	-	-	-
k	4000	16000	0.90	0.09	0.00	0.00	510990	0.08	0.04	0	569	-	-	-	-	-	-	-	-
ks	4000	16000	22.08	0.01	0.00	0.00	5763257	0.01	0.03	0	164741	-	-	-	-	-	422	-	-
ks_nopr	4000	16000	37.30	0.01	0.00	0.00	8381110	0.00	0.03	0	0	-	-	-	-	-	346848	-	-
ni	4000	16000	79.19	0.01	0.00	0.00	67906589	0.01	0.03	0	1996	-	-	-	-	1113	-	-	-
ni_nopr	4000	16000	60.60	0.01	0.00	0.00	68761057	0.01	-	-	-	-	-	-	-	1175	-	-	-
ho	4000	32000	2.08	0.13	0.00	0.00	4617361	0.13	0.07	0	2263	24	3264.51	834	875	-	-	-	-
ho_nopr	4000	32000	1.36	0.16	0.00	2.00	4158505	0.16	-	-	-	28	2762.99	2375	1595	-	-	-	-
ho_noprxs	4000	32000	1.11	0.06	0.00	2.00	3478284	0.06	-	-	-	119	904.31	3879	-	-	-	-	-
ho_noxs	4000	32000	1.50	0.04	0.00	0.00	3454252	0.04	0.06	0	2714	49	1849.89	1234	-	-	-	-	-
hybrid	4000	31941	156.91	0.01	-	-	-	-	-	-	228	-	-	-	-	-	-	-	-
k	4000	32000	1.91	0.11	0.00	0.00	938535	0.12	0.07	0	1392	-	-	-	-	-	-	-	-
ks	4000	32000	39.24	0.01	0.00	0.00	9512648	0.00	0.06	0	165141	-	-	-	-	-	360	-	-
ks_nopr	4000	32000	54.59	0.01	0.00	0.00	12258313	0.00	0.06	0	0	-	-	-	-	-	347532	-	-
ni	4000	32000	158.02	0.02	0.00	0.00	158493920	0.01	0.06	0	2172	-	-	-	-	1276	-	-	-
ni_nopr	4000	32000	123.34	0.01	0.00	0.00	161548677	0.01	-	-	-	-	-	-	-	1520	-	-	-

Table A.19: REG1 data

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	4000	64000	3.68	0.21	0.00	0.00	8894012	0.23	0.11	0	2235	25	3192.64	1132	604	-	-	-	-
ho_nopr	4000	64000	2.35	0.12	0.00	0.00	7959014	0.12	-	-	-	26	3096.85	3171	801	-	-	-	-
ho_noprxs	4000	64000	1.88	0.03	0.00	1.22	6573041	0.03	-	-	-	91	1102.45	3907	-	-	-	-	-
ho_noxs	4000	64000	2.79	0.05	0.00	0.00	6881485	0.04	0.11	0	2542	43	1906.94	1412	-	-	-	-	-
hybrid	4000	63766	275.75	0.01	-	-	-	-	-	-	465	-	-	-	-	-	-	-	-
k	4000	64000	4.18	0.09	0.00	0.00	1654546	0.12	0.12	0	1736	-	-	-	-	-	-	-	-
ks	4000	64000	75.65	0.00	0.00	0.00	17991690	0.00	0.11	0	160776	-	-	-	-	-	360	-	-
ks_nopr	4000	64000	90.53	0.00	0.00	0.00	20453291	0.00	0.11	0	0	-	-	-	-	-	346067	-	-
ni	4000	64000	284.58	0.01	0.00	2.00	313932692	0.00	0.11	0	2527	-	-	-	-	1253	-	-	-
ni_nopr	4000	64000	225.76	0.01	0.00	0.00	322722081	0.00	-	-	-	-	-	-	-	1733	-	-	-
ho	4000	128000	4.63	0.16	0.01	0.82	11374344	0.18	0.21	0	3054	17	3031.67	707	218	-	-	-	-
ho_nopr	4000	128000	3.68	0.11	0.00	1.22	12969070	0.12	-	-	-	22	2870.50	3621	355	-	-	-	-
ho_noprxs	4000	128000	3.51	0.07	0.00	0.00	13008224	0.07	-	-	-	48	1602.61	3950	-	-	-	-	-
ho_noxs	4000	128000	4.58	0.11	0.00	0.00	12012080	0.09	0.20	0	3374	33	2018.39	590	-	-	-	-	-
hybrid	4000	127011	495.89	0.01	-	-	-	-	-	-	636	-	-	-	-	-	-	-	-
k	4000	128000	10.74	0.07	0.00	0.00	3159411	0.12	0.24	0	1802	-	-	-	-	-	-	-	-
ks	4000	128000	146.24	0.00	0.00	0.00	34587947	0.00	0.20	0	158712	-	-	-	-	-	360	-	-
ks_nopr	4000	128000	161.09	0.00	0.00	0.00	36546665	0.00	0.21	0	0	-	-	-	-	-	345795	-	-
ni	4000	128000	527.19	0.01	0.00	0.00	608916828	0.00	0.21	0	2685	-	-	-	-	1215	-	-	-
ni_nopr	4000	128000	419.54	0.02	0.00	0.00	630270549	0.00	-	-	-	-	-	-	-	1863	-	-	-
ho	4000	256000	9.25	0.26	0.00	2.00	22283721	0.29	0.41	0	2827	14	3315.66	1039	117	-	-	-	-
ho_nopr	4000	256000	7.98	0.13	0.00	2.00	27625737	0.14	-	-	-	18	3216.83	3758	222	-	-	-	-
ho_noprxs	4000	256000	6.96	0.08	0.00	2.00	25427920	0.11	-	-	-	50	1385.19	3948	-	-	-	-	-
ho_noxs	4000	256000	7.72	0.11	0.00	2.00	19627500	0.12	0.40	0	3643	28	1878.71	326	-	-	-	-	-
hybrid	4000	252001	915.74	0.01	-	-	-	-	-	-	749	-	-	-	-	-	-	-	-
k	4000	256000	30.48	0.05	0.00	3.39	6116382	0.13	0.46	0	1837	-	-	-	-	-	-	-	-
ks	4000	256000	279.81	0.00	0.00	4.90	66016826	0.00	0.40	0	157742	-	-	-	-	-	360	-	-
ks_nopr	4000	256000	295.49	0.00	0.00	0.00	67213720	0.00	0.40	0	0	-	-	-	-	-	346700	-	-
ni	3199	204799	789.97	0.50	-0.20	-2.00	929500495	0.50	0.12	0	2242	-	-	-	-	935	-	-	-
ni_nopr	4000	256000	788.60	0.02	0.00	0.00	1214900453	0.00	-	-	-	-	-	-	-	1932	-	-	-
ho	4000	512000	17.68	0.20	0.00	2.00	40734493	0.22	0.81	0	3215	11	3348.59	689	81	-	-	-	-
ho_nopr	4000	512000	15.99	0.13	0.00	2.00	56315335	0.17	-	-	-	16	3311.86	3848	133	-	-	-	-
ho_noprxs	4000	512000	13.88	0.12	0.00	0.00	50086275	0.11	-	-	-	32	1783.39	3966	-	-	-	-	-
ho_noxs	4000	512000	16.44	0.12	0.00	0.00	40759551	0.12	0.79	0	3695	20	2172.18	282	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k	4000	512000	-	-	-	-	-	-	0.91	0	-	-	-	-	-	-	-	-	-
ks	4000	512000	525.44	0.00	0.00	3.39	123186585	0.00	0.80	0	156994	-	-	-	-	-	360	-	-
ks_nopr	4000	512000	545.35	0.00	0.00	0.00	123563248	0.00	0.80	0	0	-	-	-	-	-	346158	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	8000	32000	3.07	0.14	0.01	0.82	5349231	0.14	0.09	0	3043	29	6380.80	1876	3049	-	-	-	-
ho_nopr	8000	32000	1.88	0.26	0.00	0.00	4137229	0.28	-	-	-	30	5167.75	3482	4486	-	-	-	-
ho_noprxs	8000	32000	2.31	0.15	0.00	1.22	5218807	0.15	-	-	-	488	560.02	7510	-	-	-	-	-
ho_noxs	8000	32000	3.13	0.14	0.00	1.22	5612719	0.15	0.09	0	5479	68	3615.62	2450	-	-	-	-	-
hybrid	8000	31990	368.04	0.02	-	-	-	-	-	-	72	-	-	-	-	-	-	-	-
k	8000	32000	2.05	0.06	0.00	0.00	1058509	0.05	0.09	0	1135	-	-	-	-	-	-	-	-
ks	8000	32000	52.80	0.01	0.00	0.00	11897638	0.01	0.08	0	357872	-	-	-	-	-	407	-	-
ks_nopr	8000	32000	85.86	0.00	0.00	0.00	18187806	0.00	0.08	0	0	-	-	-	-	-	730797	-	-
ni	8000	32000	362.46	0.01	0.00	2.00	259302500	0.01	0.08	0	3772	-	-	-	-	2121	-	-	-
ni_nopr	8000	32000	276.21	0.01	0.00	0.00	262058041	0.01	-	-	-	-	-	-	-	2225	-	-	-
ho	1000	8000	0.24	0.08	0.00	0.00	758049	0.08	0.01	0	566	15	854.30	124	291	-	-	-	-
ho_nopr	1000	8000	0.16	0.12	0.00	0.00	701438	0.12	-	-	-	18	694.54	602	378	-	-	-	-
ho_noprxs	1000	8000	0.15	0.12	0.00	0.00	665581	0.09	-	-	-	76	230.51	922	-	-	-	-	-
ho_noxs	1000	8000	0.20	0.08	0.00	0.00	668381	0.09	0.01	0	751	29	510.06	217	-	-	-	-	-
hybrid	1000	7943	7.42	0.01	-	-	-	-	-	-	58	-	-	-	-	-	-	-	-
k	1000	8000	0.33	0.10	0.00	0.00	218697	0.11	0.02	0	345	-	-	-	-	-	-	-	-
ks	1000	8000	6.66	0.03	0.00	0.00	2078410	0.01	0.01	0	35743	-	-	-	-	-	329	-	-
ks_nopr	1000	8000	10.07	0.01	0.00	0.00	2542820	0.01	0.01	0	0	-	-	-	-	-	76668	-	-
ni	1000	8000	7.82	0.02	0.00	0.00	10471363	0.02	0.01	0	562	-	-	-	-	341	-	-	-
ni_nopr	1000	8000	6.12	0.02	0.00	0.00	10731553	0.01	-	-	-	-	-	-	-	406	-	-	-

Table A.20: REG1 data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	8000	64000	7.09	0.16	0.00	2.00	14194306	0.17	0.14	0	4316	37	6635.68	1547	2096	-	-	-	-
ho_nopr	8000	64000	3.36	0.27	0.00	1.22	8762890	0.30	-	-	-	32	5364.86	4835	3131	-	-	-	-
ho_noprxs	8000	64000	2.68	0.05	0.01	0.50	6957842	0.04	-	-	-	302	937.47	7696	-	-	-	-	-
ho_noxs	8000	64000	3.40	0.05	0.00	1.22	6840538	0.05	0.15	0	5678	52	3841.81	2267	-	-	-	-	-
hybrid	8000	63948	714.37	0.01	-	-	-	-	-	-	432	-	-	-	-	-	-	-	-
k	8000	64000	4.24	0.18	0.00	0.00	1858429	0.15	0.15	0	2802	-	-	-	-	-	-	-	-
ks	8000	64000	91.46	0.00	0.00	0.00	20025578	0.00	0.14	0	346811	-	-	-	-	-	378	-	-
ks_nopr	8000	64000	123.40	0.00	0.00	4.90	26316320	0.00	0.14	0	0	-	-	-	-	-	727433	-	-
ni	8000	64000	738.59	0.01	0.00	0.00	611253237	0.01	0.14	0	4320	-	-	-	-	2453	-	-	-
ni_nopr	8000	64000	573.43	0.01	0.00	0.00	622358458	0.01	-	-	-	-	-	-	-	2911	-	-	-
ho	8000	128000	10.87	0.15	0.01	0.82	23304030	0.15	0.26	0	5470	30	6550.52	1449	1048	-	-	-	-
ho_nopr	8000	128000	7.68	0.12	0.00	1.22	23371174	0.13	-	-	-	35	6314.17	6335	1628	-	-	-	-
ho_noprxs	8000	128000	5.18	0.09	0.00	1.22	15517549	0.11	-	-	-	145	1714.43	7853	-	-	-	-	-
ho_noxs	8000	128000	6.67	0.12	0.00	0.00	14688219	0.13	0.26	0	6034	50	3670.72	1912	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k	8000	128000	9.18	0.11	0.00	0.00	3355821	0.13	0.28	0	3486	-	-	-	-	-	-	-	-
ks	8000	128000	174.70	0.00	0.00	0.00	38135235	0.00	0.25	0	337446	-	-	-	-	-	378	-	-
ks_nopr	8000	128000	202.99	0.00	0.00	0.00	43767767	0.00	0.25	0	0	-	-	-	-	-	727152	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	8000	256000	15.28	0.25	0.01	0.50	33064404	0.27	0.49	0	6055	20	6648.82	1457	465	-	-	-	-
ho_nopr	8000	256000	11.84	0.16	0.01	0.50	35746668	0.19	-	-	-	25	6193.91	7105	869	-	-	-	-
ho_noprxs	8000	256000	8.44	0.08	0.00	2.00	26675254	0.08	-	-	-	91	2519.54	7907	-	-	-	-	-
ho_noxs	8000	256000	10.79	0.08	0.01	0.50	24461412	0.08	0.48	0	6979	40	3957.14	978	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k	8000	256000	24.34	0.09	0.00	0.00	6794776	0.15	0.53	0	3609	-	-	-	-	-	-	-	-
ks	8000	256000	338.76	0.00	0.00	0.00	74046194	0.00	0.47	0	332994	-	-	-	-	-	378	-	-
ks_nopr	8000	256000	361.45	0.00	0.00	0.00	78586178	0.00	0.47	0	0	-	-	-	-	-	727495	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	8000	512000	28.43	0.20	0.01	0.82	61742367	0.21	0.96	0	6497	20	6479.37	1282	199	-	-	-	-
ho_nopr	8000	512000	22.58	0.11	0.00	2.00	71228873	0.13	-	-	-	24	6310.14	7635	339	-	-	-	-
ho_noprxs	8000	512000	18.72	0.06	0.00	1.22	57552572	0.06	-	-	-	64	3211.41	7934	-	-	-	-	-
ho_noxs	8000	512000	21.18	0.08	0.00	2.00	46832524	0.08	0.94	0	7186	35	4358.37	776	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k	8000	512000	-	-	-	-	-	-	1.04	0	-	-	-	-	-	-	-	-	-
ks	8000	512000	658.49	0.00	0.00	0.00	143623202	0.00	0.92	0	331050	-	-	-	-	-	378	-	-
ks_nopr	8000	512000	672.61	0.00	0.00	0.00	146332256	0.00	0.92	0	0	-	-	-	-	-	726871	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	8000	1024000	51.22	0.24	0.01	0.82	106495621	0.27	1.90	0	7080	17	6462.20	748	151	-	-	-	-
ho_nopr	8000	1024000	40.60	0.10	0.00	0.00	125669240	0.14	-	-	-	23	6055.29	7636	339	-	-	-	-
ho_noprxs	8000	1024000	34.54	0.10	0.01	0.82	106160539	0.10	-	-	-	43	3099.51	7955	-	-	-	-	-
ho_noxs	8000	1024000	40.97	0.15	0.01	0.50	87924075	0.15	1.90	0	7624	25	3989.45	347	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k	8000	1024000	-	-	-	-	-	-	2.10	0	-	-	-	-	-	-	-	-	-
ks	8000	1024000	-	-	-	-	-	-	1.86	0	-	-	-	-	-	-	-	-	-
ks_nopr	8000	1024000	-	-	-	-	-	-	1.86	0	-	-	-	-	-	-	-	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	16000	64000	6.90	0.44	0.01	0.00	11112970	0.46	0.20	0	7094	32	12468.52	3085	5786	-	-	-	-
ho_nopr	16000	64000	3.37	0.12	0.01	0.00	6569316	0.12	-	-	-	30	9769.41	7103	8865	-	-	-	-
ho_noprxs	16000	64000	5.53	0.24	0.01	0.82	11248730	0.24	-	-	-	1069	580.03	14930	-	-	-	-	-
ho_noxs	16000	64000	7.25	0.11	0.01	0.50	12035648	0.11	0.19	0	12053	77	7054.88	3866	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k	16000	64000	4.56	0.09	0.00	0.00	2221004	0.08	0.19	0	2282	-	-	-	-	-	-	-	-
ks	16000	64000	126.62	0.00	0.00	0.00	24658153	0.00	0.18	0	765278	-	-	-	-	-	403	-	-
ks_nopr	16000	64000	194.56	0.00	0.00	0.00	39090134	0.00	0.18	0	0	-	-	-	-	-	1526418	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.21: REG1 data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	16000	128000	17.11	0.30	0.01	0.00	31875748	0.31	0.32	0	8916	41	13020.00	3372	3667	-	-	-	-
ho_nopr	16000	128000	10.61	0.27	0.01	0.82	25654192	0.29	-	-	-	41	11670.50	9545	6412	-	-	-	-
ho_noprxs	16000	128000	7.16	0.07	0.01	0.50	17475261	0.08	-	-	-	534	1053.18	15464	-	-	-	-	-
ho_noxs	16000	128000	9.38	0.03	0.01	0.50	17926654	0.03	0.32	0	10881	65	7254.96	5051	-	-	-	-	-
k	16000	128000	9.16	0.16	0.00	4.90	3791428	0.14	0.33	0	5562	-	-	-	-	-	-	-	-
ks	16000	128000	216.40	0.00	0.00	0.00	42000477	0.00	0.30	0	726276	-	-	-	-	-	396	-	-
ks_nopr	16000	128000	278.11	0.00	0.00	0.00	56276091	0.00	0.30	0	0	-	-	-	-	-	1522991	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	16000	256000	35.24	0.40	0.01	0.00	71527009	0.42	0.58	0	11122	43	13495.44	2735	2096	-	-	-	-
ho_nopr	16000	256000	18.80	0.19	0.01	0.50	51019905	0.22	-	-	-	37	12371.29	12211	3750	-	-	-	-
ho_noprxs	16000	256000	11.73	0.06	0.01	0.50	31955553	0.06	-	-	-	263	2795.22	15736	-	-	-	-	-
ho_noxs	16000	256000	16.07	0.08	0.01	0.00	32860069	0.09	0.57	0	11436	60	7176.72	4500	-	-	-	-	-
k	16000	256000	19.57	0.08	0.00	0.00	6696886	0.11	0.60	0	6954	-	-	-	-	-	-	-	-
ks	16000	256000	409.20	0.00	0.00	0.00	80340882	0.00	0.54	0	707313	-	-	-	-	-	396	-	-
ks_nopr	16000	256000	455.98	0.00	0.00	0.00	93254279	0.00	0.54	0	0	-	-	-	-	-	1523218	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	16000	512000	44.02	0.15	0.01	0.00	90563234	0.17	1.10	0	13434	29	13384.69	1562	971	-	-	-	-
ho_nopr	16000	512000	32.89	0.12	0.01	0.50	93592364	0.14	-	-	-	35	12221.72	14148	1815	-	-	-	-
ho_noprxs	16000	512000	22.37	0.08	0.01	0.82	64322918	0.08	-	-	-	104	4732.34	15894	-	-	-	-	-
ho_noxs	16000	512000	29.36	0.07	0.01	0.50	62855621	0.08	1.07	0	11882	50	7559.74	4065	-	-	-	-	-
k	16000	512000	-	-	-	-	-	-	1.16	0	-	-	-	-	-	-	-	-	-
ks	16000	512000	795.45	0.00	0.00	4.90	156929724	0.00	1.04	0	698021	-	-	-	-	-	396	-	-
ks_nopr	16000	512000	814.96	0.00	0.00	0.00	167381430	0.00	1.04	0	0	-	-	-	-	-	1524793	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	16000	1024000	63.34	0.15	0.01	0.00	128573917	0.18	2.13	0	12767	19	12708.32	2880	331	-	-	-	-
ho_nopr	16000	1024000	56.20	0.12	0.01	0.50	155440655	0.14	-	-	-	25	11917.58	15164	809	-	-	-	-
ho_noprxs	16000	1024000	52.54	0.11	0.01	0.82	154857878	0.13	-	-	-	63	6336.16	15935	-	-	-	-	-
ho_noxs	16000	1024000	65.22	0.15	0.01	0.50	139855022	0.16	2.12	0	12554	45	7641.91	3397	-	-	-	-	-
k	16000	1024000	-	-	-	-	-	-	2.30	0	-	-	-	-	-	-	-	-	-
ks	16000	1024000	-	-	-	-	-	-	2.07	0	-	-	-	-	-	-	-	-	-
ks_nopr	16000	1024000	-	-	-	-	-	-	2.08	0	-	-	-	-	-	-	-	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	1000	16000	0.41	0.18	0.00	0.00	1299160	0.18	0.02	0	684	13	833.78	163	137	-	-	-	-
ho_nopr	1000	16000	0.32	0.07	0.00	0.00	1359336	0.09	-	-	-	16	736.16	789	192	-	-	-	-
ho_noprxs	1000	16000	0.29	0.06	0.00	0.00	1280009	0.06	-	-	-	51	297.10	947	-	-	-	-	-
ho_noxs	1000	16000	0.38	0.04	0.00	0.00	1244717	0.07	0.03	0	678	24	517.94	295	-	-	-	-	-
hybrid	1000	15771	13.65	0.01	-	-	-	-	-	-	117	-	-	-	-	-	-	-	-
k	1000	16000	0.91	0.11	0.00	0.00	457862	0.14	0.03	0	433	-	-	-	-	-	-	-	-
ks	1000	16000	13.57	0.01	0.00	0.00	3781221	0.01	0.02	0	35168	-	-	-	-	-	319	-	-
ks_nopr	1000	16000	17.09	0.01	0.00	0.00	4225565	0.01	0.02	0	0	-	-	-	-	-	76974	-	-
ni	1000	16000	14.97	0.01	0.00	0.00	20029919	0.01	0.02	0	647	-	-	-	-	326	-	-	-
ni_nopr	1000	16000	11.59	0.01	0.00	2.00	20744438	0.01	-	-	-	-	-	-	-	451	-	-	-
ho	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho_noprxs	16000	2048000	90.84	0.10	0.01	0.82	251801003	0.08	-	-	-	62	5997.49	15936	-	-	-	-	-
ho_noxs	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ks	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ks_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ho	1000	32000	0.75	0.06	0.00	0.00	2327044	0.06	0.05	0	803	10	766.77	128	55	-	-	-	-
ho_nopr	1000	32000	0.62	0.08	0.00	0.00	2644521	0.07	-	-	-	15	688.23	867	116	-	-	-	-
ho_noprxs	1000	32000	0.57	0.05	0.00	0.00	2461502	0.05	-	-	-	36	362.37	962	-	-	-	-	-
ho_noxs	1000	32000	0.68	0.09	0.00	0.00	2186642	0.08	0.04	0	810	20	525.48	168	-	-	-	-	-
hybrid	1000	31030	25.94	0.01	-	-	-	-	-	-	153	-	-	-	-	-	-	-	-
k	1000	32000	2.37	0.09	0.00	0.00	874052	0.14	0.05	0	450	-	-	-	-	-	-	-	-
ks	1000	32000	26.89	0.01	0.00	0.00	6981176	0.01	0.04	0	35008	-	-	-	-	-	318	-	-
ks_nopr	1000	32000	30.39	0.01	0.00	0.00	7357750	0.01	0.04	0	0	-	-	-	-	-	76783	-	-
ni	1000	32000	27.35	0.01	0.00	0.00	37359850	0.00	0.04	0	675	-	-	-	-	310	-	-	-
ni_nopr	1000	32000	21.26	0.00	0.00	0.00	39060234	0.00	-	-	-	-	-	-	-	476	-	-	-

Table A.22: REG1 data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	1000	64000	1.24	0.10	0.00	0.00	3766219	0.11	0.09	0	714	9	891.37	237	37	-	-	-	-
ho_nopr	1000	64000	1.24	0.11	0.00	0.00	5152551	0.13	-	-	-	14	811.01	930	55	-	-	-	-
ho_noprxs	1000	64000	1.24	0.08	0.00	0.00	5201893	0.06	-	-	-	25	464.35	973	-	-	-	-	-
ho_noxs	1000	64000	1.27	0.06	0.00	0.00	3921484	0.05	0.09	0	757	14	600.66	225	-	-	-	-	-
hybrid	1000	60150	47.73	0.01	-	-	-	-	-	-	182	-	-	-	-	-	-	-	-
k	1000	64000	6.53	0.05	0.00	0.00	1578973	0.16	0.10	0	457	-	-	-	-	-	-	-	-
ks	1000	64000	49.52	0.01	0.00	0.00	12519128	0.01	0.08	0	34727	-	-	-	-	-	318	-	-
ks_nopr	1000	64000	53.55	0.01	0.00	4.90	12835589	0.01	0.08	0	0	-	-	-	-	-	76948	-	-
ni	1000	64000	50.86	0.01	0.00	0.00	68021712	0.01	0.08	0	700	-	-	-	-	295	-	-	-
ni_nopr	1000	64000	39.52	0.01	0.00	0.00	72026238	0.00	-	-	-	-	-	-	-	487	-	-	-
ho	1000	128000	2.32	0.04	0.00	0.00	6504475	0.04	0.16	0	854	7	929.49	123	13	-	-	-	-
ho_nopr	1000	128000	2.46	0.11	0.00	0.00	9501742	0.10	-	-	-	12	817.10	961	25	-	-	-	-
ho_noprxs	1000	128000	2.40	0.11	0.00	0.00	9389644	0.10	-	-	-	23	448.98	975	-	-	-	-	-
ho_noxs	1000	128000	2.12	0.06	0.00	0.00	6038684	0.05	0.16	0	964	11	569.76	22	-	-	-	-	-
hybrid	1000	113035	84.20	0.00	-	-	-	-	-	-	199	-	-	-	-	-	-	-	-
k	1000	128000	34.27	0.02	0.00	2.29	27284014	0.03	0.18	0	467	-	-	-	-	-	-	2.32	29.86
ks	1000	128000	86.25	0.00	0.00	4.90	21477712	0.01	0.16	0	34709	-	-	-	-	-	318	-	-
ks_nopr	1000	128000	91.63	0.01	0.00	3.39	21715768	0.01	0.16	0	0	-	-	-	-	-	76807	-	-
ni	1000	128000	95.13	0.01	0.00	0.00	119028361	0.01	0.16	0	709	-	-	-	-	283	-	-	-
ni_nopr	1000	128000	72.43	0.01	0.00	0.00	127300992	0.00	-	-	-	-	-	-	-	495	-	-	-
ho	2000	8000	0.27	0.14	0.00	0.00	659042	0.14	0.02	0	822	14	1660.26	352	808	-	-	-	-
ho_nopr	2000	8000	0.22	0.16	0.00	0.00	711731	0.18	-	-	-	21	1309.04	825	1151	-	-	-	-
ho_noprxs	2000	8000	0.29	0.07	0.00	0.00	941397	0.08	-	-	-	152	327.39	1846	-	-	-	-	-
ho_noxs	2000	8000	0.36	0.12	0.00	0.00	907043	0.12	0.02	0	1438	41	965.91	517	-	-	-	-	-
hybrid	2000	7986	17.25	0.02	-	-	-	-	-	-	24	-	-	-	-	-	-	-	-
k	2000	8000	0.41	0.11	0.00	0.00	263932	0.10	0.02	0	289	-	-	-	-	-	-	-	-
ks	2000	8000	8.64	0.02	0.00	0.00	2716837	0.02	0.01	0	75221	-	-	-	-	-	418	-	-
ks_nopr	2000	8000	15.88	0.01	0.00	0.00	3785550	0.01	0.01	0	0	-	-	-	-	-	163060	-	-
ni	2000	8000	16.70	0.02	0.00	0.00	17444423	0.02	0.01	0	911	-	-	-	-	573	-	-	-
ni_nopr	2000	8000	12.80	0.02	0.00	0.00	17645731	0.02	-	-	-	-	-	-	-	612	-	-	-
ho	2000	16000	0.64	0.18	0.00	0.00	1668922	0.19	0.03	0	852	18	1599.61	617	510	-	-	-	-
ho_nopr	2000	16000	0.45	0.29	0.00	0.00	1599719	0.31	-	-	-	28	1127.13	1180	790	-	-	-	-
ho_noprxs	2000	16000	0.43	0.11	0.00	2.00	1562451	0.11	-	-	-	108	490.18	1891	-	-	-	-	-
ho_noxs	2000	16000	0.58	0.08	0.00	0.00	1591363	0.08	0.03	0	1201	41	1000.46	755	-	-	-	-	-
hybrid	2000	15947	35.29	0.01	-	-	-	-	-	-	113	-	-	-	-	-	-	-	-
k	2000	16000	0.81	0.08	0.00	0.00	445916	0.08	0.03	0	694	-	-	-	-	-	-	-	-
ks	2000	16000	16.16	0.01	0.00	0.00	4412345	0.00	0.02	0	76505	-	-	-	-	-	339	-	-
ks_nopr	2000	16000	23.68	0.01	0.00	4.90	5592339	0.00	0.03	0	0	-	-	-	-	-	162757	-	-
ni	2000	16000	35.44	0.00	0.00	0.00	40906159	0.01	0.03	0	1156	-	-	-	-	661	-	-	-
ni_nopr	2000	16000	27.92	0.02	0.00	0.00	41915701	0.01	-	-	-	-	-	-	-	794	-	-	-
ho	2000	32000	1.22	0.24	0.00	0.00	3378001	0.25	0.05	0	1380	16	1685.77	311	290	-	-	-	-
ho_nopr	2000	32000	0.86	0.18	0.00	0.00	3383194	0.21	-	-	-	18	1602.60	1626	353	-	-	-	-
ho_noprxs	2000	32000	0.73	0.07	0.00	0.00	2922104	0.07	-	-	-	56	667.52	1942	-	-	-	-	-
ho_noxs	2000	32000	1.05	0.09	0.00	0.00	3020974	0.09	0.05	0	1327	33	960.52	636	-	-	-	-	-
hybrid	2000	31759	63.07	0.01	-	-	-	-	-	-	235	-	-	-	-	-	-	-	-
k	2000	32000	1.80	0.02	0.00	0.00	769758	0.01	0.06	0	867	-	-	-	-	-	-	-	-
ks	2000	32000	32.30	0.00	0.00	0.00	8244158	0.00	0.05	0	74912	-	-	-	-	-	336	-	-
ks_nopr	2000	32000	39.52	0.00	0.00	0.00	9284073	0.00	0.05	0	0	-	-	-	-	-	161739	-	-
ni	2000	32000	64.37	0.01	0.00	0.00	79999226	0.00	0.05	0	1275	-	-	-	-	643	-	-	-
ni_nopr	2000	32000	49.68	0.01	0.00	2.00	82416061	0.00	-	-	-	-	-	-	-	886	-	-	-

Table A.23: REG1 data (cont)

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	128	1024	0.01	0.33	0.00	0.00	43225	0.09	0.00	0	66	7	104.66	24	27	-	-	-	-
ho_nopr	128	1024	0.01	0.00	0.00	0.00	44064	0.11	-	-	-	11	86.61	67	47	-	-	-	-
ho_noprxs	128	1024	0.01	0.00	0.00	0.00	47845	0.06	-	-	-	21	51.14	105	-	-	-	-	-
ho_noxs	128	1024	0.01	0.33	0.00	0.00	49986	0.08	0.00	0	62	13	70.68	50	-	-	-	-	-
hybrid	128	967	0.10	0.04	-	-	-	-	-	-	4	-	-	-	-	-	-	-	-
k	128	1024	0.02	0.33	0.00	0.00	24547	0.01	0.00	0	41	-	-	-	-	-	-	-	-
ks	128	1024	0.52	0.02	0.00	0.00	191230	0.02	0.00	0	3647	-	-	-	-	-	284	-	-
ks_nopr	128	1024	0.82	0.04	0.00	0.00	216848	0.03	0.00	0	0	-	-	-	-	-	8100	-	-
ni	128	1024	0.08	0.05	0.00	0.00	161684	0.03	0.00	0	78	-	-	-	-	44	-	-	-
ni_nopr	128	1024	0.07	0.06	0.00	0.00	174641	0.03	-	-	-	-	-	-	-	56	-	-	-
ho	256	4096	0.05	0.11	0.00	0.00	213928	0.06	0.01	0	145	7	219.42	66	34	-	-	-	-
ho_nopr	256	4096	0.04	0.14	0.00	0.00	225677	0.04	-	-	-	12	172.11	190	53	-	-	-	-
ho_noprxs	256	4096	0.04	0.14	0.00	0.00	231942	0.11	-	-	-	26	99.52	228	-	-	-	-	-
ho_noxs	256	4096	0.05	0.15	0.00	0.00	219728	0.10	0.00	0	172	14	156.15	67	-	-	-	-	-
hybrid	256	3859	0.67	0.03	-	-	-	-	-	-	26	-	-	-	-	-	-	-	-
k	256	4096	0.13	0.12	0.00	0.00	106149	0.14	0.00	0	109	-	-	-	-	-	-	-	-
ks	256	4096	2.12	0.02	0.00	0.00	739686	0.02	0.00	0	7620	-	-	-	-	-	289	-	-
ks_nopr	256	4096	2.95	0.02	0.00	0.00	808518	0.01	0.00	0	0	-	-	-	-	-	16938	-	-
ni	256	4096	0.66	0.06	0.00	0.00	1255611	0.01	0.00	0	160	-	-	-	-	86	-	-	-
ni_nopr	256	4096	0.52	0.03	0.00	0.00	1323981	0.01	-	-	-	-	-	-	-	118	-	-	-
ho	512	16384	0.26	0.03	0.00	2.00	835486	0.07	0.02	0	374	6	476.33	98	31	-	-	-	-
ho_nopr	512	16384	0.25	0.06	0.00	2.00	1106118	0.05	-	-	-	10	415.47	445	55	-	-	-	-
ho_noprxs	512	16384	0.25	0.18	0.00	0.00	1146966	0.19	-	-	-	26	215.08	484	-	-	-	-	-
ho_noxs	512	16384	0.29	0.14	0.00	0.00	993982	0.14	0.02	0	372	15	283.98	121	-	-	-	-	-
hybrid	512	15424	5.68	0.01	-	-	-	-	-	-	76	-	-	-	-	-	-	-	-
k	512	16384	0.99	0.06	0.00	4.90	384460	0.13	0.03	0	230	-	-	-	-	-	-	-	-
ks	512	16384	10.99	0.01	0.00	0.00	3091358	0.01	0.02	0	16669	-	-	-	-	-	303	-	-
ks_nopr	512	16384	12.74	0.01	0.00	4.90	3242684	0.01	0.02	0	0	-	-	-	-	-	37000	-	-
ni	512	16384	6.65	0.00	0.00	0.00	9260699	0.00	0.02	0	349	-	-	-	-	157	-	-	-
ni_nopr	512	16384	5.11	0.01	0.00	0.00	9886507	0.01	-	-	-	-	-	-	-	246	-	-	-
ho	1024	65536	1.25	0.12	0.00	0.00	3740683	0.13	0.09	0	759	9	862.93	218	34	-	-	-	-
ho_nopr	1024	65536	1.21	0.07	0.00	0.00	4971482	0.07	-	-	-	13	796.36	945	63	-	-	-	-
ho_noprxs	1024	65536	1.19	0.10	0.00	0.00	4960265	0.09	-	-	-	26	474.56	996	-	-	-	-	-
ho_noxs	1024	65536	1.35	0.14	0.00	0.00	4158297	0.15	0.09	0	792	16	583.88	213	-	-	-	-	-
hybrid	1024	61688	50.06	0.00	-	-	-	-	-	-	187	-	-	-	-	-	-	-	-
k	1024	65536	6.71	0.05	0.00	0.00	1616941	0.16	0.10	0	467	-	-	-	-	-	-	-	-
ks	1024	65536	51.04	0.01	0.00	0.00	12833888	0.01	0.08	0	35461	-	-	-	-	-	318	-	-
ks_nopr	1024	65536	55.32	0.01	0.00	0.00	13236592	0.01	0.08	0	0	-	-	-	-	-	79103	-	-
ni	1024	65536	54.00	0.01	0.00	0.00	71774491	0.00	0.08	0	713	-	-	-	-	303	-	-	-
ni_nopr	1024	65536	41.75	0.01	0.00	0.00	75764030	0.00	-	-	-	-	-	-	-	500	-	-	-
ho	2048	262144	6.27	0.19	0.00	1.22	16791158	0.22	0.35	0	1735	9	1688.60	263	37	-	-	-	-
ho_nopr	2048	262144	6.29	0.11	0.00	0.00	23640286	0.13	-	-	-	14	1655.01	1972	61	-	-	-	-
ho_noprxs	2048	262144	5.97	0.14	0.00	0.00	22739058	0.11	-	-	-	26	943.23	2020	-	-	-	-	-
ho_noxs	2048	262144	6.26	0.13	0.00	0.00	17264754	0.13	0.36	0	1611	14	1198.45	419	-	-	-	-	-
hybrid	2048	246520	398.25	0.01	-	-	-	-	-	-	418	-	-	-	-	-	-	-	-
k	2048	262144	96.37	0.02	0.00	0.00	72228454	0.02	0.41	0	957	-	-	-	-	-	-	5.85	85.76
ks	2048	262144	222.06	0.00	0.00	0.00	53837716	0.00	0.36	0	75056	-	-	-	-	-	336	-	-
ks_nopr	2048	262144	232.81	0.00	0.00	0.00	54115840	0.00	0.36	0	0	-	-	-	-	-	164886	-	-
ni	2048	262144	434.57	0.01	0.00	2.00	547706475	0.01	0.36	0	1462	-	-	-	-	578	-	-	-
ni_nopr	2048	262144	340.20	0.01	0.00	0.00	581596560	0.00	-	-	-	-	-	-	-	1011	-	-	-

Table A.24: REG2 data

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	4000	16000	1.10	0.27	0.00	0.00	2218968	0.28	0.04	0	2040	23	3419.16	633	1301	-	-	-	-
ho_nopr	4000	16000	0.75	0.34	0.00	1.22	1909436	0.36	-	-	-	28	2609.32	1680	2290	-	-	-	-
ho_noprxs	4000	16000	0.84	0.13	0.00	0.00	2244400	0.14	-	-	-	323	347.81	3675	-	-	-	-	-
ho_noxs	4000	16000	1.14	0.11	0.00	0.00	2355196	0.11	0.04	0	2623	53	1817.30	1321	-	-	-	-	-
hybrid	4000	15985	79.18	0.02	-	-	-	-	-	-	45	-	-	-	-	-	-	-	-
k	4000	16000	0.89	0.03	0.00	0.00	501402	0.03	0.04	0	567	-	-	-	-	-	-	-	-
ks	4000	16000	22.08	0.02	0.00	0.00	5753487	0.01	0.03	0	165006	-	-	-	-	-	421	-	-
ks_nopr	4000	16000	37.59	0.01	0.00	0.00	8388216	0.00	0.04	0	0	-	-	-	-	-	347384	-	-
ni	4000	16000	77.86	0.02	0.00	0.00	66914214	0.02	0.04	0	1770	-	-	-	-	1096	-	-	-
ni_nopr	4000	16000	59.35	0.02	0.00	0.00	67717242	0.01	-	-	-	-	-	-	-	1163	-	-	-
ho	4000	17872	0.43	0.11	0.00	2.00	842669	0.10	0.04	0	945	10	3886.56	774	2267	-	-	-	-
ho_nopr	4000	17872	0.34	0.09	0.00	1.22	834081	0.08	-	-	-	13	3307.72	1393	2592	-	-	-	-
ho_noprxs	4000	17872	0.76	0.05	0.00	1.22	2031613	0.05	-	-	-	220	524.08	3778	-	-	-	-	-
ho_noxs	4000	17872	1.05	0.11	0.00	2.00	2205910	0.11	0.04	0	2803	52	2064.47	1141	-	-	-	-	-
hybrid	4000	17854	3.17	0.01	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	17872	0.79	0.15	0.00	0.00	443427	0.09	0.04	0	426	-	-	-	-	-	-	-	-
ks	4000	17872	20.29	0.01	0.00	0.00	5035862	0.00	0.04	0	143979	-	-	-	-	-	360	-	-
ks_nopr	4000	17872	27.28	0.01	0.00	4.90	6284075	0.00	0.04	0	0	-	-	-	-	-	143359	-	-
ni	4000	17872	2.23	0.01	0.00	0.00	1954958	0.00	0.04	0	1373	-	-	-	-	28	-	-	-
ni_nopr	4000	17872	1.71	0.01	0.00	0.00	2007138	0.01	-	-	-	-	-	-	-	29	-	-	-
ho	4000	17968	0.42	0.09	0.00	0.00	831186	0.10	0.04	0	928	10	3839.83	387	2672	-	-	-	-
ho_nopr	4000	17968	0.35	0.05	0.00	0.00	830606	0.05	-	-	-	11	3506.69	1275	2711	-	-	-	-
ho_noprxs	4000	17968	0.78	0.07	0.00	2.00	2095834	0.07	-	-	-	182	639.74	3816	-	-	-	-	-
ho_noxs	4000	17968	1.06	0.07	0.00	2.00	2220416	0.07	0.04	0	2796	52	2019.31	1149	-	-	-	-	-
hybrid	4000	17949	2.82	0.04	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	17968	0.71	0.03	0.00	3.39	411508	0.00	0.04	0	425	-	-	-	-	-	-	-	-
ks	4000	17968	20.33	0.01	0.00	4.90	5032291	0.00	0.04	0	142452	-	-	-	-	-	360	-	-
ks_nopr	4000	17968	26.85	0.01	0.00	2.71	6200325	0.00	0.04	0	0	-	-	-	-	-	134606	-	-
ni	4000	17968	1.96	0.04	0.00	0.00	1729860	0.06	0.03	0	901	-	-	-	-	24	-	-	-
ni_nopr	4000	17968	1.51	0.02	0.00	0.00	1784856	0.01	-	-	-	-	-	-	-	26	-	-	-
ho	4000	17992	0.43	0.09	0.00	2.00	835761	0.11	0.04	0	1206	12	3847.51	445	2334	-	-	-	-
ho_nopr	4000	17992	0.35	0.10	0.00	2.00	858993	0.10	-	-	-	13	3606.06	1379	2606	-	-	-	-
ho_noprxs	4000	17992	0.80	0.10	0.00	0.00	2186246	0.10	-	-	-	229	576.78	3769	-	-	-	-	-
ho_noxs	4000	17992	1.05	0.18	0.00	0.00	2216035	0.19	0.04	0	2612	55	2164.63	1330	-	-	-	-	-
hybrid	4000	17973	2.92	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	17992	0.71	0.02	0.00	3.39	411881	0.00	0.05	0	436	-	-	-	-	-	-	-	-
ks	4000	17992	20.35	0.01	0.00	4.90	5032954	0.00	0.04	0	142124	-	-	-	-	-	360	-	-
ks_nopr	4000	17992	26.76	0.01	0.00	4.90	6175910	0.00	0.04	0	0	-	-	-	-	-	131979	-	-
ni	4000	17992	1.92	0.02	0.00	0.00	1705051	0.02	0.04	0	1408	-	-	-	-	24	-	-	-
ni_nopr	4000	17992	1.49	0.02	0.00	0.00	1765166	0.02	-	-	-	-	-	-	-	26	-	-	-
ho	4000	17998	0.43	0.11	0.01	0.50	877556	0.12	0.04	0	791	13	3896.46	674	2518	-	-	-	-
ho_nopr	4000	17998	0.33	0.10	0.00	1.22	813494	0.10	-	-	-	16	3498.00	1192	2790	-	-	-	-
ho_noprxs	4000	17998	0.74	0.03	0.00	2.00	2006488	0.04	-	-	-	185	661.52	3813	-	-	-	-	-
ho_noxs	4000	17998	1.10	0.09	0.01	0.00	2320532	0.09	0.04	0	2735	54	2074.15	1208	-	-	-	-	-
hybrid	4000	17979	2.92	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	17998	0.71	0.02	0.01	0.71	411903	0.00	0.04	0	431	-	-	-	-	-	-	-	-
ks	4000	17998	20.30	0.01	0.01	0.89	5030813	0.00	0.04	0	142064	-	-	-	-	-	360	-	-
ks_nopr	4000	17998	26.68	0.01	0.00	1.13	6168150	0.00	0.04	0	0	-	-	-	-	-	131548	-	-
ni	4000	17998	1.90	0.03	0.00	1.22	1655569	0.03	0.04	0	1265	-	-	-	-	23	-	-	-
ni_nopr	4000	17998	1.44	0.02	0.00	1.22	1714658	0.01	-	-	-	-	-	-	-	25	-	-	-
ho	4000	17999	0.43	0.12	0.01	0.63	841826	0.14	0.04	0	1195	11	3764.74	357	2433	-	-	-	-
ho_nopr	4000	17999	0.35	0.06	0.01	0.82	849377	0.05	-	-	-	13	3502.78	1340	2645	-	-	-	-
ho_noprxs	4000	17999	0.81	0.20	0.01	0.33	2208973	0.20	-	-	-	193	626.92	3806	-	-	-	-	-
ho_noxs	4000	17999	0.96	0.07	0.01	0.63	2025369	0.07	0.04	0	3033	49	2014.70	915	-	-	-	-	-
hybrid	4000	17980	2.88	0.05	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	17999	0.71	0.03	0.02	0.47	411794	0.00	0.04	0	420	-	-	-	-	-	-	-	-
ks	4000	17999	20.38	0.01	0.01	0.87	5031412	0.00	0.04	0	142035	-	-	-	-	-	360	-	-
ks_nopr	4000	17999	26.73	0.01	0.01	0.64	6168597	0.00	0.04	0	0	-	-	-	-	-	130641	-	-
ni	4000	17999	1.90	0.01	0.01	0.63	1679990	0.00	0.04	0	837	-	-	-	-	24	-	-	-
ni_nopr	4000	17999	1.45	0.01	0.01	0.94	1709862	0.01	-	-	-	-	-	-	-	25	-	-	-

Table A.25: IRREG data

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	4000	18000	1.25	0.35	0.00	2.00	2562969	0.36	0.04	0	1513	29	3276.71	1203	1251	-	-	-	-
ho_nopr	4000	18000	0.78	0.30	0.00	0.00	2078615	0.32	-	-	-	91	1006.76	2309	1597	-	-	-	-
ho_noprxs	4000	18000	0.80	0.06	0.00	1.22	2161851	0.05	-	-	-	164	704.74	3834	-	-	-	-	-
ho_noxs	4000	18000	1.10	0.05	0.00	0.00	2312688	0.05	0.04	0	2160	54	1849.36	1782	-	-	-	-	-
hybrid	4000	17981	91.23	0.01	-	-	-	-	-	-	67	-	-	-	-	-	-	-	-
k	4000	18000	1.05	0.10	0.00	0.00	575287	0.09	0.05	0	1142	-	-	-	-	-	-	-	-
ks	4000	18000	24.41	0.01	0.00	0.00	6173769	0.01	0.04	0	168068	-	-	-	-	-	390	-	-
ks_nopr	4000	18000	40.94	0.01	0.00	0.00	8820460	0.00	0.04	0	0	-	-	-	-	-	346083	-	-
ni	4000	18000	89.22	0.01	0.00	0.00	80311388	0.01	0.04	0	1683	-	-	-	-	1166	-	-	-
ni_nopr	4000	18000	68.90	0.01	0.00	0.00	81207069	0.01	-	-	-	-	-	-	-	1242	-	-	-
ho	4000	18000	0.93	0.20	0.00	0.00	1901201	0.21	0.04	0	1647	21	3155.41	1035	1294	-	-	-	-
ho_nopr	4000	18000	0.67	0.25	0.00	0.00	1763930	0.27	-	-	-	79	956.99	2308	1611	-	-	-	-
ho_noprxs	4000	18000	0.77	0.11	0.00	1.22	2122690	0.10	-	-	-	194	626.68	3805	-	-	-	-	-
ho_noxs	4000	18000	1.09	0.10	0.00	2.00	2282047	0.10	0.04	0	2233	57	1894.28	1707	-	-	-	-	-
hybrid	4000	17981	91.42	0.01	-	-	-	-	-	-	62	-	-	-	-	-	-	-	-
k	4000	18000	1.07	0.10	0.00	0.00	590160	0.10	0.04	0	1145	-	-	-	-	-	-	-	-
ks	4000	18000	24.50	0.01	0.00	0.00	6193166	0.01	0.04	0	167707	-	-	-	-	-	394	-	-
ks_nopr	4000	18000	40.81	0.01	0.00	0.00	8828381	0.00	0.04	0	0	-	-	-	-	-	347322	-	-
ni	4000	18000	89.24	0.00	0.00	0.00	80224341	0.00	0.04	0	1974	-	-	-	-	1165	-	-	-
ni_nopr	4000	18000	69.38	0.01	0.00	2.00	81191613	0.00	-	-	-	-	-	-	-	1244	-	-	-
ho	4000	18001	1.14	0.27	0.00	2.00	2329625	0.28	0.04	0	1657	26	3252.60	1069	1244	-	-	-	-
ho_nopr	4000	18001	0.84	0.22	0.00	2.00	2228460	0.23	-	-	-	92	1018.24	2256	1649	-	-	-	-
ho_noprxs	4000	18001	0.87	0.10	0.00	2.00	2365175	0.11	-	-	-	149	718.70	3850	-	-	-	-	-
ho_noxs	4000	18001	1.19	0.10	0.00	2.00	2495374	0.10	0.04	0	2411	56	1761.70	1531	-	-	-	-	-
hybrid	4000	17981	91.68	0.01	-	-	-	-	-	-	64	-	-	-	-	-	-	-	-
k	4000	18001	0.99	0.03	0.00	0.00	542349	0.01	0.05	0	1143	-	-	-	-	-	-	-	-
ks	4000	18001	24.43	0.01	0.00	0.00	6180522	0.01	0.04	0	168158	-	-	-	-	-	390	-	-
ks_nopr	4000	18001	40.96	0.01	0.00	0.00	8823849	0.00	0.04	0	0	-	-	-	-	-	347629	-	-
ni	4000	18001	90.10	0.02	0.00	0.00	80000674	0.01	0.04	0	2013	-	-	-	-	1162	-	-	-
ni_nopr	4000	18001	69.31	0.01	0.00	0.00	81014302	0.01	-	-	-	-	-	-	-	1243	-	-	-
ho	4000	18002	1.06	0.14	0.00	0.00	2155042	0.15	0.04	0	1285	22	3153.64	1353	1337	-	-	-	-
ho_nopr	4000	18002	0.81	0.38	0.00	0.00	2126929	0.40	-	-	-	88	914.88	2285	1624	-	-	-	-
ho_noprxs	4000	18002	0.75	0.05	0.00	0.00	2033741	0.03	-	-	-	224	474.26	3775	-	-	-	-	-
ho_noxs	4000	18002	1.09	0.06	0.00	0.00	2285722	0.06	0.04	0	2186	55	1783.70	1756	-	-	-	-	-
hybrid	4000	17982	90.26	0.01	-	-	-	-	-	-	69	-	-	-	-	-	-	-	-
k	4000	18002	1.12	0.16	0.00	0.00	607685	0.13	0.04	0	1129	-	-	-	-	-	-	-	-
ks	4000	18002	24.34	0.01	0.00	4.90	6169723	0.01	0.04	0	168557	-	-	-	-	-	387	-	-
ks_nopr	4000	18002	41.13	0.01	0.00	0.00	8827699	0.01	0.04	0	0	-	-	-	-	-	346908	-	-
ni	4000	18002	91.30	0.02	0.00	0.00	80456815	0.00	0.04	0	1752	-	-	-	-	1168	-	-	-
ni_nopr	4000	18002	69.36	0.01	0.00	0.00	81306040	0.00	-	-	-	-	-	-	-	1242	-	-	-
ho	4000	18008	1.07	0.32	0.00	0.00	2182638	0.33	0.04	0	1719	24	3189.11	912	1342	-	-	-	-
ho_nopr	4000	18008	0.66	0.17	0.00	0.00	1763623	0.17	-	-	-	78	935.97	2349	1571	-	-	-	-
ho_noprxs	4000	18008	0.78	0.03	0.00	1.22	2097683	0.03	-	-	-	182	640.76	3816	-	-	-	-	-
ho_noxs	4000	18008	1.18	0.13	0.00	2.00	2458874	0.13	0.04	0	2414	56	1927.10	1527	-	-	-	-	-
hybrid	4000	17988	91.04	0.02	-	-	-	-	-	-	55	-	-	-	-	-	-	-	-
k	4000	18008	1.10	0.09	0.00	0.00	606997	0.08	0.04	0	1125	-	-	-	-	-	-	-	-
ks	4000	18008	24.37	0.01	0.00	0.00	6171061	0.01	0.04	0	168314	-	-	-	-	-	388	-	-
ks_nopr	4000	18008	40.77	0.01	0.00	0.00	8814297	0.00	0.04	0	0	-	-	-	-	-	346420	-	-
ni	4000	18008	89.82	0.01	0.00	0.00	80028251	0.01	0.04	0	1884	-	-	-	-	1161	-	-	-
ni_nopr	4000	18008	68.76	0.01	0.00	0.00	80896637	0.01	-	-	-	-	-	-	-	1237	-	-	-
ho	4000	16001	1.27	0.17	0.00	2.00	2594228	0.18	0.04	0	2235	28	3307.98	443	1291	-	-	-	-
ho_nopr	4000	16001	0.61	0.18	0.00	0.00	1532746	0.19	-	-	-	26	2507.27	1723	2249	-	-	-	-
ho_noprxs	4000	16001	0.76	0.10	0.00	0.00	2010436	0.09	-	-	-	250	456.10	3749	-	-	-	-	-
ho_noxs	4000	16001	1.07	0.10	0.00	1.22	2220403	0.11	0.04	0	2309	54	1837.07	1634	-	-	-	-	-
hybrid	4000	15986	78.62	0.01	-	-	-	-	-	-	45	-	-	-	-	-	-	-	-
k	4000	16001	0.87	0.02	0.00	0.00	488465	0.02	0.04	0	574	-	-	-	-	-	-	-	-
ks	4000	16001	22.09	0.01	0.00	0.00	5766984	0.01	0.03	0	164683	-	-	-	-	-	424	-	-
ks_nopr	4000	16001	37.47	0.00	0.00	0.00	8387672	0.00	0.03	0	0	-	-	-	-	-	347468	-	-
ni	4000	16001	77.57	0.01	0.00	0.00	67447020	0.01	0.03	0	1785	-	-	-	-	1105	-	-	-
ni_nopr	4000	16001	58.91	0.01	0.00	0.00	68170275	0.01	-	-	-	-	-	-	-	1169	-	-	-

Table A.26: IRREG data (cont)

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	4000	18032	0.96	0.12	0.00	0.00	1984185	0.13	0.04	0	1286	22	3197.95	1317	1371	-	-	-	-
ho_nopr	4000	18032	0.66	0.07	0.00	0.00	1740324	0.08	-	-	-	68	1091.26	2342	1589	-	-	-	-
ho_noprxs	4000	18032	0.77	0.10	0.00	0.00	2050222	0.10	-	-	-	210	538.28	3788	-	-	-	-	-
ho_noxs	4000	18032	1.08	0.11	0.00	1.22	2250851	0.11	0.04	0	2472	54	1882.78	1471	-	-	-	-	-
hybrid	4000	18012	89.10	0.01	-	-	-	-	-	-	45	-	-	-	-	-	-	-	-
k	4000	18032	0.97	0.02	0.00	0.00	538513	0.01	0.04	0	1138	-	-	-	-	-	-	-	-
ks	4000	18032	24.37	0.01	0.00	0.00	6165349	0.01	0.04	0	167755	-	-	-	-	-	390	-	-
ks_nopr	4000	18032	40.64	0.01	0.00	0.00	8793752	0.00	0.04	0	0	-	-	-	-	-	344086	-	-
ni	4000	18032	87.33	0.01	0.00	0.00	78176303	0.02	0.04	0	1846	-	-	-	-	1132	-	-	-
ni_nopr	4000	18032	67.44	0.02	0.00	0.00	78925101	0.02	-	-	-	-	-	-	-	1194	-	-	-
ho	4000	18128	0.88	0.17	0.00	0.00	1776700	0.17	0.04	0	1332	23	3073.79	1310	1331	-	-	-	-
ho_nopr	4000	18128	0.71	0.23	0.00	2.00	1884016	0.24	-	-	-	69	1187.47	2240	1689	-	-	-	-
ho_noprxs	4000	18128	0.80	0.11	0.00	2.00	2183758	0.11	-	-	-	196	569.18	3803	-	-	-	-	-
ho_noxs	4000	18128	1.26	0.06	0.00	0.00	2645618	0.07	0.04	0	2149	57	1835.22	1791	-	-	-	-	-
hybrid	4000	18108	81.77	0.02	-	-	-	-	-	-	16	-	-	-	-	-	-	-	-
k	4000	18128	1.07	0.10	0.00	0.00	581624	0.09	0.05	0	1110	-	-	-	-	-	-	-	-
ks	4000	18128	24.09	0.01	0.00	0.00	6076427	0.01	0.04	0	167700	-	-	-	-	-	378	-	-
ks_nopr	4000	18128	39.86	0.01	0.00	0.00	8670435	0.00	0.04	0	0	-	-	-	-	-	333896	-	-
ni	4000	18128	80.20	0.01	0.00	0.00	71326768	0.01	0.04	0	1659	-	-	-	-	1024	-	-	-
ni_nopr	4000	18128	60.79	0.01	0.00	0.00	71695319	0.01	-	-	-	-	-	-	-	1054	-	-	-
ho	4000	18512	0.69	0.16	0.00	2.00	1385153	0.16	0.04	0	2146	18	3533.07	579	1254	-	-	-	-
ho_nopr	4000	18512	0.53	0.19	0.00	0.00	1419907	0.21	-	-	-	57	1211.71	2068	1873	-	-	-	-
ho_noprxs	4000	18512	0.80	0.07	0.00	0.00	2205651	0.07	-	-	-	255	466.80	3744	-	-	-	-	-
ho_noxs	4000	18512	1.18	0.09	0.00	2.00	2488518	0.10	0.04	0	2542	52	2049.84	1403	-	-	-	-	-
hybrid	4000	18491	53.07	0.00	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	18512	1.04	0.08	0.00	0.00	569064	0.08	0.05	0	1049	-	-	-	-	-	-	-	-
ks	4000	18512	23.43	0.01	0.00	0.00	5842193	0.00	0.04	0	164379	-	-	-	-	-	362	-	-
ks_nopr	4000	18512	37.55	0.01	0.00	0.00	8201296	0.00	0.04	0	0	-	-	-	-	-	292333	-	-
ni	4000	18512	51.78	0.01	0.00	0.00	45398091	0.01	0.04	0	1724	-	-	-	-	632	-	-	-
ni_nopr	4000	18512	38.83	0.01	0.00	2.00	45468836	0.01	-	-	-	-	-	-	-	638	-	-	-
ho	4000	19000	0.49	0.04	0.00	2.00	972028	0.05	0.04	0	2025	11	3731.31	382	1578	-	-	-	-
ho_nopr	4000	19000	0.47	0.09	0.00	1.22	1217452	0.12	-	-	-	86	627.65	1386	2526	-	-	-	-
ho_noprxs	4000	19000	0.85	0.02	0.00	2.00	2334362	0.02	-	-	-	256	437.38	3743	-	-	-	-	-
ho_noxs	4000	19000	0.96	0.06	0.00	2.00	2041341	0.07	0.05	0	2703	46	1771.97	1248	-	-	-	-	-
hybrid	4000	18978	24.25	0.03	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	19000	1.03	0.03	0.00	0.00	554985	0.01	0.04	0	1000	-	-	-	-	-	-	-	-
ks	4000	19000	23.26	0.01	0.00	0.00	5685895	0.00	0.04	0	157747	-	-	-	-	-	360	-	-
ks_nopr	4000	19000	34.99	0.01	0.00	0.00	7693147	0.00	0.04	0	0	-	-	-	-	-	242427	-	-
ni	4000	19000	23.27	0.03	0.00	0.00	20368629	0.03	0.04	0	1124	-	-	-	-	274	-	-	-
ni_nopr	4000	19000	17.45	0.03	0.00	0.00	20407409	0.03	-	-	-	-	-	-	-	276	-	-	-
ho	4000	19488	0.50	0.11	0.00	0.00	1011545	0.12	0.04	0	1726	8	3819.50	446	1817	-	-	-	-
ho_nopr	4000	19488	0.42	0.09	0.00	2.00	1066158	0.07	-	-	-	30	1397.73	1316	2651	-	-	-	-
ho_noprxs	4000	19488	0.89	0.12	0.00	0.00	2484925	0.14	-	-	-	295	415.25	3703	-	-	-	-	-
ho_noxs	4000	19488	1.24	0.11	0.00	0.00	2639453	0.11	0.04	0	2657	53	2028.45	1287	-	-	-	-	-
hybrid	4000	19465	8.20	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	19488	1.06	0.02	0.00	0.00	551010	0.01	0.05	0	942	-	-	-	-	-	-	-	-
ks	4000	19488	23.17	0.01	0.00	0.00	5583931	0.00	0.04	0	150606	-	-	-	-	-	360	-	-
ks_nopr	4000	19488	32.57	0.01	0.00	0.00	7222560	0.00	0.04	0	0	-	-	-	-	-	193009	-	-
ni	4000	19488	7.17	0.02	0.00	0.00	6277318	0.03	0.04	0	563	-	-	-	-	82	-	-	-
ni_nopr	4000	19488	5.36	0.03	0.00	2.00	6298579	0.03	-	-	-	-	-	-	-	83	-	-	-
ho	4000	19872	0.45	0.07	0.00	0.00	904165	0.08	0.04	0	1359	10	3721.60	463	2165	-	-	-	-
ho_nopr	4000	19872	0.37	0.07	0.00	0.00	952205	0.06	-	-	-	13	3160.31	1163	2822	-	-	-	-
ho_noprxs	4000	19872	0.82	0.07	0.00	0.00	2254566	0.07	-	-	-	233	516.35	3765	-	-	-	-	-
ho_noxs	4000	19872	1.01	0.09	0.00	2.00	2162225	0.09	0.04	0	3145	50	2027.03	801	-	-	-	-	-
hybrid	4000	19847	4.06	0.01	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	19872	0.96	0.13	0.00	0.00	513041	0.10	0.05	0	910	-	-	-	-	-	-	-	-
ks	4000	19872	23.11	0.01	0.00	4.90	5544245	0.00	0.04	0	145232	-	-	-	-	-	360	-	-
ks_nopr	4000	19872	30.83	0.01	0.00	0.00	6883541	0.00	0.04	0	0	-	-	-	-	-	156283	-	-
ni	4000	19872	3.14	0.01	0.00	0.00	2806724	0.01	0.04	0	961	-	-	-	-	36	-	-	-
ni_nopr	4000	19872	2.38	0.01	0.00	0.00	2844712	0.01	-	-	-	-	-	-	-	37	-	-	-

Table A.27: IRREG data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	4000	19968	0.51	0.03	0.00	2.00	1024664	0.02	0.04	0	1160	11	3943.26	530	2295	-	-	-	-
ho_nopr	4000	19968	0.38	0.06	0.00	2.00	976694	0.04	-	-	-	15	3467.80	1201	2782	-	-	-	-
ho_noprxs	4000	19968	0.84	0.13	0.00	2.00	2345380	0.14	-	-	-	269	438.74	3730	-	-	-	-	-
ho_noxs	4000	19968	1.06	0.11	0.00	2.00	2292255	0.11	0.04	0	2794	52	2013.51	1151	-	-	-	-	-
hybrid	4000	19942	3.68	0.03	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	19968	0.86	0.11	0.00	4.90	474495	0.07	0.05	0	902	-	-	-	-	-	-	-	-
ks	4000	19968	23.18	0.01	0.00	0.00	5533196	0.00	0.04	0	143928	-	-	-	-	-	360	-	-
ks_nopr	4000	19968	30.54	0.01	0.00	4.90	6813107	0.00	0.04	0	0	-	-	-	-	-	147222	-	-
ni	4000	19968	2.71	0.01	0.00	0.00	2476826	0.00	0.04	0	631	-	-	-	-	32	-	-	-
ni_nopr	4000	19968	2.08	0.01	0.00	0.00	2500252	0.01	-	-	-	-	-	-	-	33	-	-	-
ho	4000	19992	0.47	0.11	0.00	0.00	947404	0.14	0.04	0	1265	14	3873.20	312	2406	-	-	-	-
ho_nopr	4000	19992	0.39	0.06	0.00	1.22	982945	0.05	-	-	-	15	3717.79	1232	2751	-	-	-	-
ho_noprxs	4000	19992	0.82	0.06	0.00	2.00	2254118	0.07	-	-	-	234	464.57	3764	-	-	-	-	-
ho_noxs	4000	19992	1.05	0.12	0.00	1.22	2229070	0.12	0.04	0	2879	47	2016.02	1070	-	-	-	-	-
hybrid	4000	19966	3.60	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	19992	0.81	0.02	0.00	1.46	458854	0.00	0.05	0	894	-	-	-	-	-	-	-	-
ks	4000	19992	23.25	0.01	0.00	2.29	5533477	0.00	0.04	0	143650	-	-	-	-	-	360	-	-
ks_nopr	4000	19992	30.46	0.01	0.00	1.78	6796199	0.00	0.04	0	0	-	-	-	-	-	144886	-	-
ni	4000	19992	2.53	0.07	0.00	0.00	2292778	0.09	0.04	0	1230	-	-	-	-	29	-	-	-
ni_nopr	4000	19992	2.01	0.02	0.00	1.22	2419368	0.02	-	-	-	-	-	-	-	32	-	-	-
ho	4000	19998	0.50	0.11	0.01	0.63	1021437	0.12	0.05	0	1239	12	3859.26	444	2302	-	-	-	-
ho_nopr	4000	19998	0.37	0.10	0.00	1.22	950583	0.07	-	-	-	15	3306.01	1052	2930	-	-	-	-
ho_noprxs	4000	19998	0.79	0.10	0.01	0.33	2210902	0.09	-	-	-	243	472.28	3756	-	-	-	-	-
ho_noxs	4000	19998	1.14	0.20	0.01	0.94	2450897	0.20	0.04	0	2939	48	2148.75	1010	-	-	-	-	-
hybrid	4000	19972	3.67	0.03	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	19998	0.90	0.13	0.01	0.67	496252	0.09	0.05	0	892	-	-	-	-	-	-	-	-
ks	4000	19998	23.21	0.01	0.01	1.00	5534196	0.00	0.04	0	143547	-	-	-	-	-	360	-	-
ks_nopr	4000	19998	30.46	0.01	0.01	0.99	6791941	0.00	0.04	0	0	-	-	-	-	-	144367	-	-
ni	4000	19998	2.53	0.05	0.01	0.94	2282411	0.05	0.04	0	2006	-	-	-	-	29	-	-	-
ni_nopr	4000	19998	2.00	0.01	0.01	0.89	2418235	0.00	-	-	-	-	-	-	-	32	-	-	-
ho	4000	19999	0.47	0.06	0.01	0.63	937796	0.06	0.04	0	1134	12	3861.75	283	2567	-	-	-	-
ho_nopr	4000	19999	0.37	0.06	0.01	0.33	942599	0.05	-	-	-	15	3579.70	1144	2839	-	-	-	-
ho_noprxs	4000	19999	0.84	0.12	0.01	0.82	2386909	0.13	-	-	-	217	587.49	3781	-	-	-	-	-
ho_noxs	4000	19999	1.08	0.08	0.01	0.63	2295285	0.08	0.04	0	2970	52	2030.25	975	-	-	-	-	-
hybrid	4000	19973	3.60	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	19999	0.84	0.09	0.01	0.45	477199	0.08	0.05	0	905	-	-	-	-	-	-	-	-
ks	4000	19999	23.13	0.01	0.01	0.62	5532939	0.00	0.04	0	143532	-	-	-	-	-	360	-	-
ks_nopr	4000	19999	30.30	0.00	0.01	0.75	6791292	0.00	0.04	0	0	-	-	-	-	-	144935	-	-
ni	4000	19999	2.64	0.02	0.01	0.89	2387316	0.03	0.04	0	994	-	-	-	-	30	-	-	-
ni_nopr	4000	19999	2.01	0.02	0.01	0.63	2449209	0.01	-	-	-	-	-	-	-	32	-	-	-
ho	4000	16002	0.83	0.35	0.00	0.00	1652344	0.37	0.04	0	1808	18	3186.96	681	1490	-	-	-	-
ho_nopr	4000	16002	0.52	0.09	0.00	0.00	1330925	0.11	-	-	-	23	2205.47	1636	2339	-	-	-	-
ho_noprxs	4000	16002	0.73	0.05	0.00	0.00	1918943	0.06	-	-	-	319	302.91	3679	-	-	-	-	-
ho_noxs	4000	16002	1.11	0.10	0.00	2.00	2256278	0.11	0.04	0	2671	54	1705.83	1272	-	-	-	-	-
hybrid	4000	15987	79.26	0.02	-	-	-	-	-	-	46	-	-	-	-	-	-	-	-
k	4000	16002	0.87	0.02	0.00	0.00	492495	0.01	0.04	0	565	-	-	-	-	-	-	-	-
ks	4000	16002	22.25	0.02	0.00	0.00	5780443	0.01	0.03	0	163947	-	-	-	-	-	428	-	-
ks_nopr	4000	16002	37.44	0.01	0.00	0.00	8382547	0.00	0.03	0	0	-	-	-	-	-	347803	-	-
ni	4000	16002	77.92	0.01	0.00	0.00	67221757	0.01	0.03	0	1718	-	-	-	-	1101	-	-	-
ni_nopr	4000	16002	59.35	0.01	0.00	0.00	67936727	0.01	-	-	-	-	-	-	-	1166	-	-	-
ho	4000	20000	1.26	0.21	0.00	2.00	2650657	0.22	0.04	0	1981	25	3254.40	702	1289	-	-	-	-
ho_nopr	4000	20000	1.00	0.26	0.00	1.22	2816372	0.28	-	-	-	33	2741.98	2037	1928	-	-	-	-
ho_noprxs	4000	20000	0.85	0.08	0.00	1.22	2383670	0.08	-	-	-	229	506.78	3769	-	-	-	-	-
ho_noxs	4000	20000	1.36	0.12	0.00	0.00	2920179	0.12	0.04	0	2733	57	1768.63	1207	-	-	-	-	-
hybrid	4000	19974	103.02	0.01	-	-	-	-	-	-	85	-	-	-	-	-	-	-	-
k	4000	20000	1.24	0.17	0.00	4.90	659768	0.14	0.05	0	904	-	-	-	-	-	-	-	-
ks	4000	20000	26.04	0.01	0.00	0.00	6487797	0.01	0.04	0	169330	-	-	-	-	-	367	-	-
ks_nopr	4000	20000	42.81	0.01	0.00	0.00	9311809	0.00	0.04	0	0	-	-	-	-	-	348464	-	-
ni	4000	20000	102.33	0.01	0.00	0.00	92641778	0.01	0.04	0	1950	-	-	-	-	1207	-	-	-
ni_nopr	4000	20000	77.86	0.02	0.00	0.00	93762348	0.01	-	-	-	-	-	-	-	1305	-	-	-

Table A.28: IRREG data (cont)

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	edge scans avg	edge scans dev %	preprocess time	initial PR	internal PR	s-t cuts	avg. size	1 node layers	excess detect	phases	leaves	packing time	respect time
ho	4000	16008	1.07	0.33	0.00	2.00	2162918	0.34	0.04	0	1843	23	3403.26	618	1513	-	-	-	-
ho_nopr	4000	16008	0.60	0.30	0.00	1.22	1534529	0.33	-	-	-	25	2514.46	1817	2156	-	-	-	-
ho_noprxs	4000	16008	0.85	0.08	0.00	1.22	2201395	0.08	-	-	-	269	409.25	3729	-	-	-	-	-
ho_noxs	4000	16008	1.08	0.09	0.00	2.00	2191507	0.10	0.04	0	2750	55	1739.21	1192	-	-	-	-	-
hybrid	4000	15993	78.13	0.01	-	-	-	-	-	-	43	-	-	-	-	-	-	-	-
k	4000	16008	0.92	0.11	0.00	0.00	513722	0.09	0.04	0	566	-	-	-	-	-	-	-	-
ks	4000	16008	22.00	0.01	0.00	0.00	5743012	0.01	0.03	0	165318	-	-	-	-	-	419	-	-
ks_nopr	4000	16008	37.34	0.01	0.00	0.00	8367371	0.00	0.04	0	0	-	-	-	-	-	346167	-	-
ni	4000	16008	76.40	0.02	0.00	0.00	66466780	0.01	0.04	0	1833	-	-	-	-	1088	-	-	-
ni_nopr	4000	16008	58.80	0.01	0.00	0.00	67086529	0.01	-	-	-	-	-	-	-	1141	-	-	-
ho	4000	16032	1.04	0.15	0.00	0.00	2093962	0.16	0.04	0	2060	25	3315.68	518	1393	-	-	-	-
ho_nopr	4000	16032	0.68	0.20	0.00	2.00	1752914	0.20	-	-	-	34	2349.00	1570	2394	-	-	-	-
ho_noprxs	4000	16032	0.80	0.08	0.00	0.00	2111676	0.08	-	-	-	290	417.26	3709	-	-	-	-	-
ho_noxs	4000	16032	1.12	0.13	0.00	0.00	2287736	0.13	0.04	0	2687	57	1888.02	1253	-	-	-	-	-
hybrid	4000	16017	77.70	0.01	-	-	-	-	-	-	31	-	-	-	-	-	-	-	-
k	4000	16032	0.87	0.03	0.00	0.00	497553	0.03	0.04	0	565	-	-	-	-	-	-	-	-
ks	4000	16032	21.92	0.01	0.00	0.00	5725190	0.01	0.03	0	165271	-	-	-	-	-	416	-	-
ks_nopr	4000	16032	37.30	0.01	0.00	0.00	8348505	0.00	0.04	0	0	-	-	-	-	-	343248	-	-
ni	4000	16032	74.51	0.01	0.00	0.00	64753799	0.01	0.03	0	1712	-	-	-	-	1058	-	-	-
ni_nopr	4000	16032	57.05	0.01	0.00	0.00	65349654	0.01	-	-	-	-	-	-	-	1112	-	-	-
ho	4000	16128	1.07	0.20	0.00	0.00	2171971	0.21	0.04	0	2090	24	3351.14	454	1429	-	-	-	-
ho_nopr	4000	16128	0.72	0.31	0.00	0.00	1850099	0.32	-	-	-	44	1698.74	1586	2368	-	-	-	-
ho_noprxs	4000	16128	0.77	0.11	0.00	0.00	2017220	0.12	-	-	-	265	406.14	3733	-	-	-	-	-
ho_noxs	4000	16128	1.15	0.12	0.00	0.00	2353055	0.12	0.04	0	3048	58	1842.28	891	-	-	-	-	-
hybrid	4000	16113	70.16	0.01	-	-	-	-	-	-	14	-	-	-	-	-	-	-	-
k	4000	16128	0.95	0.11	0.00	0.00	523967	0.09	0.04	0	559	-	-	-	-	-	-	-	-
ks	4000	16128	21.61	0.01	0.00	0.00	5621301	0.01	0.04	0	166054	-	-	-	-	-	399	-	-
ks_nopr	4000	16128	36.61	0.01	0.00	0.00	8209925	0.00	0.04	0	0	-	-	-	-	-	333561	-	-
ni	4000	16128	68.69	0.02	0.00	0.00	58722033	0.02	0.04	0	1733	-	-	-	-	950	-	-	-
ni_nopr	4000	16128	51.64	0.02	0.00	0.00	59036711	0.02	-	-	-	-	-	-	-	978	-	-	-
ho	4000	16512	0.54	0.13	0.00	1.22	1039862	0.13	0.04	0	2057	13	3579.16	561	1366	-	-	-	-
ho_nopr	4000	16512	0.50	0.11	0.00	2.00	1246168	0.12	-	-	-	47	1269.31	1463	2487	-	-	-	-
ho_noprxs	4000	16512	0.78	0.16	0.00	1.22	2064607	0.16	-	-	-	332	372.36	3666	-	-	-	-	-
ho_noxs	4000	16512	1.08	0.07	0.00	1.22	2219092	0.07	0.04	0	3017	55	1863.49	925	-	-	-	-	-
hybrid	4000	16497	44.80	0.03	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	16512	0.98	0.09	0.00	0.00	531138	0.10	0.04	0	516	-	-	-	-	-	-	-	-
ks	4000	16512	20.63	0.01	0.00	4.90	5343115	0.01	0.03	0	164679	-	-	-	-	-	367	-	-
ks_nopr	4000	16512	34.20	0.01	0.00	0.00	7708141	0.00	0.04	0	0	-	-	-	-	-	289769	-	-
ni	4000	16512	43.16	0.02	0.00	0.00	36716479	0.01	0.03	0	1223	-	-	-	-	573	-	-	-
ni_nopr	4000	16512	32.66	0.02	0.00	0.00	36758173	0.01	-	-	-	-	-	-	-	579	-	-	-
ho	4000	17000	0.44	0.06	0.00	0.00	859576	0.07	0.04	0	1884	11	3569.09	377	1724	-	-	-	-
ho_nopr	4000	17000	0.39	0.11	0.00	1.22	983675	0.13	-	-	-	60	864.45	1171	2766	-	-	-	-
ho_noprxs	4000	17000	0.74	0.04	0.00	1.22	1977996	0.04	-	-	-	283	411.09	3715	-	-	-	-	-
ho_noxs	4000	17000	1.01	0.08	0.00	0.00	2084487	0.08	0.04	0	2925	52	1843.45	1021	-	-	-	-	-
hybrid	4000	16983	19.50	0.01	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	17000	0.94	0.04	0.00	0.00	501469	0.03	0.04	0	479	-	-	-	-	-	-	-	-
ks	4000	17000	20.35	0.01	0.00	4.90	5165804	0.00	0.03	0	157875	-	-	-	-	-	360	-	-
ks_nopr	4000	17000	31.47	0.00	0.00	0.00	7131610	0.00	0.04	0	0	-	-	-	-	-	234823	-	-
ni	4000	17000	17.93	0.03	0.00	0.00	15374207	0.02	0.03	0	1644	-	-	-	-	231	-	-	-
ni_nopr	4000	17000	13.58	0.02	0.00	0.00	15431955	0.02	-	-	-	-	-	-	-	233	-	-	-
ho	4000	17488	0.43	0.10	0.00	1.22	836407	0.11	0.04	0	1548	10	3697.10	515	1923	-	-	-	-
ho_nopr	4000	17488	0.36	0.11	0.00	1.22	873946	0.10	-	-	-	23	1890.64	1242	2733	-	-	-	-
ho_noprxs	4000	17488	0.78	0.03	0.00	1.22	2083656	0.04	-	-	-	292	413.61	3706	-	-	-	-	-
ho_noxs	4000	17488	1.04	0.04	0.00	2.00	2179022	0.04	0.04	0	2945	53	1974.74	999	-	-	-	-	-
hybrid	4000	17471	6.01	0.02	-	-	-	-	-	-	0	-	-	-	-	-	-	-	-
k	4000	17488	0.95	0.03	0.00	4.90	510673	0.01	0.04	0	443	-	-	-	-	-	-	-	-
ks	4000	17488	20.31	0.01	0.00	0.00	5081639	0.00	0.04	0	150186	-	-	-	-	-	360	-	-
ks_nopr	4000	17488	29.00	0.01	0.00	0.00	6625589	0.00	0.03	0	0	-	-	-	-	-	182557	-	-
ni	4000	17488	5.09	0.02	0.00	0.00	4398846	0.03	0.04	0	1018	-	-	-	-	64	-	-	-
ni_nopr	4000	17488	3.90	0.03	0.00	2.00	4435175	0.03	-	-	-	-	-	-	-	66	-	-	-

Table A.29: IRREG data (cont)

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	1024	2045	0.11	0.00	0.00	0.00	532343	0.00	0.00	0	10	9	912.33	988	15	-	-	-	-
ho_nopr	1024	2045	0.12	0.00	0.00	0.00	699088	0.00	-	-	-	10	922.70	997	16	-	-	-	-
ho_noprxs	1024	2045	0.13	0.00	0.00	0.00	700046	0.00	-	-	-	13	790.38	1010	-	-	-	-	-
ho_noxs	1024	2045	0.12	0.00	0.00	0.00	548614	0.00	0.00	0	5	11	791.64	1007	-	-	-	-	-
hybrid	1024	2045	4.16	0.00	-	-	-	-	-	-	22	-	-	-	-	-	-	-	-
k	1024	2045	1.86	0.38	0.00	0.00	1864218	0.34	0.00	0	1	-	-	-	-	-	-	1.14	0.70
ks	1024	2045	18.43	0.03	0.00	0.00	7646982	0.03	0.00	0	237274	-	-	-	-	-	13481	-	-
ks_nopr	1024	2045	134.90	0.02	0.00	0.00	25812267	0.01	0.00	0	0	-	-	-	-	-	3422384	-	-
ni	1024	2045	1.51	0.00	0.00	0.00	2191859	0.00	0.00	0	631	-	-	-	-	324	-	-	-
ni_nopr	1024	2045	1.53	0.00	0.00	0.00	3172420	0.00	-	-	-	-	-	-	-	767	-	-	-
ho	2048	4093	0.78	0.00	0.00	0.00	2777922	0.00	0.01	0	11	10	1845.30	2008	17	-	-	-	-
ho_nopr	2048	4093	0.78	0.00	0.00	0.00	3610925	0.00	-	-	-	11	1862.91	2018	18	-	-	-	-
ho_noprxs	2048	4093	0.73	0.00	0.00	0.00	3236001	0.00	-	-	-	22	1437.09	2025	-	-	-	-	-
ho_noxs	2048	4093	0.84	0.00	0.00	0.00	3029748	0.00	0.01	0	10	22	1289.82	2015	-	-	-	-	-
hybrid	2048	4093	17.14	0.00	-	-	-	-	-	-	117	-	-	-	-	-	-	-	-
k	2048	4093	4.85	0.52	0.00	0.00	3865549	0.37	0.01	0	1	-	-	-	-	-	-	2.96	1.83
ks	2048	4093	64.20	0.02	0.00	0.00	25978941	0.02	0.01	0	761936	-	-	-	-	-	33006	-	-
ks_nopr	2048	4093	510.54	0.01	0.00	0.00	97493172	0.01	0.01	0	0	-	-	-	-	-	12985750	-	-
ni	2048	4093	7.33	0.00	0.00	0.00	8752715	0.00	0.01	0	1263	-	-	-	-	648	-	-	-
ni_nopr	2048	4093	7.36	0.00	0.00	0.00	12683080	0.00	-	-	-	-	-	-	-	1535	-	-	-
ho	4096	8189	3.40	0.00	0.00	0.00	10132262	0.00	0.03	0	15	11	3725.73	4048	20	-	-	-	-
ho_nopr	4096	8189	3.18	0.00	0.00	0.00	13101365	0.00	-	-	-	12	3755.75	4063	20	-	-	-	-
ho_noprxs	4096	8189	3.12	0.00	0.00	0.00	11023325	0.00	-	-	-	31	2364.48	4064	-	-	-	-	-
ho_noxs	4096	8189	3.52	0.00	0.00	0.00	10769201	0.00	0.03	0	10	23	2670.61	4062	-	-	-	-	-
hybrid	4096	8189	77.48	0.00	-	-	-	-	-	-	318	-	-	-	-	-	-	-	-
k	4096	8189	21.76	0.47	0.00	0.00	16827559	0.43	0.03	0	1	-	-	-	-	-	-	15.78	5.81
ks	4096	8189	217.29	0.01	0.00	0.00	88894755	0.01	0.02	0	2488045	-	-	-	-	-	82202	-	-
ks_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni	4096	8189	41.46	0.00	0.00	0.00	34922585	0.00	0.02	0	2539	-	-	-	-	1292	-	-	-
ni_nopr	4096	8189	37.55	0.00	0.00	0.00	50697637	0.00	-	-	-	-	-	-	-	3071	-	-	-
ho	8192	16381	15.84	0.00	0.01	0.00	36584820	0.00	0.05	0	13	12	7511.42	8144	21	-	-	-	-
ho_nopr	8192	16381	17.41	0.00	0.00	0.00	48378276	0.00	-	-	-	13	7562.92	8156	22	-	-	-	-
ho_noprxs	8192	16381	16.73	0.00	0.00	0.00	49471274	0.00	-	-	-	28	5680.36	8163	-	-	-	-	-
ho_noxs	8192	16381	16.49	0.00	0.00	0.00	39850260	0.00	0.06	0	12	28	5215.14	8151	-	-	-	-	-
hybrid	8192	16381	355.87	0.00	-	-	-	-	-	-	763	-	-	-	-	-	-	-	-
k	8192	16381	54.64	0.35	0.00	0.00	50494020	0.25	0.05	0	1	-	-	-	-	-	-	25.85	28.44
ks	8192	16381	772.43	0.02	0.00	0.00	307373061	0.01	0.05	0	8271666	-	-	-	-	-	217031	-	-
ks_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni	8192	16381	197.90	0.00	0.00	0.00	139427922	0.00	0.04	0	5085	-	-	-	-	2578	-	-	-
ni_nopr	8192	16381	190.70	0.00	0.00	0.00	202931617	0.00	-	-	-	-	-	-	-	6143	-	-	-
ho	16384	32765	37.84	0.00	0.01	0.00	93892240	0.00	0.11	0	14	13	15125.77	16332	23	-	-	-	-
ho_nopr	16384	32765	55.69	0.00	0.00	0.00	144763771	0.00	-	-	-	14	15214.79	16345	24	-	-	-	-
ho_noprxs	16384	32765	55.36	0.00	0.00	0.00	142891464	0.00	-	-	-	29	11558.97	16354	-	-	-	-	-
ho_noxs	16384	32765	35.91	0.00	0.01	0.00	83557492	0.00	0.11	0	10	24	11504.25	16349	-	-	-	-	-
hybrid	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
k	16384	32765	-	-	-	-	-	-	0.11	0	-	-	-	-	-	-	-	-	-
ks	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ks_nopr	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
ni	16384	32765	869.31	0.00	0.00	0.00	556928579	0.00	0.11	0	10190	-	-	-	-	5146	-	-	-
ni_nopr	16384	32765	872.64	0.00	0.00	0.00	810984322	0.00	-	-	-	-	-	-	-	12287	-	-	-
ho	32768	65533	300.16	0.00	0.02	0.00	559776502	0.00	0.23	0	15	14	30429.50	32712	25	-	-	-	-
ho_nopr	32768	65533	325.74	0.00	0.01	0.00	667965736	0.00	-	-	-	16	29185.19	32723	28	-	-	-	-
ho_noprxs	32768	65533	360.98	0.00	0.01	0.00	755676525	0.00	-	-	-	34	22560.59	32733	-	-	-	-	-
ho_noxs	32768	65533	305.72	0.00	0.02	0.00	579486594	0.00	0.23	0	14	34	21025.65	32719	-	-	-	-	-
k	32768	65533	-	-	-	-	-	-	0.23	0	-	-	-	-	-	-	-	-	-
ks	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Table A.30: BIKEWHE data

	nodes	arcs	total time		discovery time		edge scans		preprocess	initial	internal	s-t	avg.	1 node	excess	phases	leaves	packing	respect
			avg	dev %	avg	dev %	avg	dev %	time	PR	PR	cuts	size	layers	detect			time	time
ho	1024	2048	0.35	0.00	0.34	0.00	1230702	0.00	0.00	0	255	257	264.00	2	509	-	-	-	-
ho_nopr	1024	2048	0.30	0.00	0.30	0.00	1222959	0.00	-	-	-	762	259.03	4	257	-	-	-	-
ho_noprxs	1024	2048	0.31	0.00	0.31	0.00	1223465	0.00	-	-	-	1019	257.77	4	-	-	-	-	-
ho_noxs	1024	2048	0.37	0.00	0.34	0.00	1228466	0.00	0.00	0	513	509	256.60	1	-	-	-	-	-
hybrid	1024	2048	0.41	0.00	-	-	-	-	-	-	660	-	-	-	-	-	-	-	-
k	1024	2048	0.29	0.08	0.01	0.82	248286	0.10	0.01	0	510	-	-	-	-	-	-	0.15	0.12
ks	1024	2048	1.11	0.02	0.05	0.51	628211	0.02	0.00	0	38488	-	-	-	-	-	503	-	-
ks_nopr	128	256	12.33	0.03	3.22	0.45	1792355	0.03	0.00	0	0	-	-	-	-	-	488208	-	-
ni	1024	2048	0.02	0.00	0.01	0.00	28666	0.00	0.01	0	1018	-	-	-	-	4	-	-	-
ni_nopr	1024	2048	1.66	0.00	0.00	0.00	4198347	0.00	-	-	-	-	-	-	-	1022	-	-	-
ho	2048	4096	1.59	0.00	1.58	0.00	4792474	0.00	0.01	0	512	513	520.00	1	1021	-	-	-	-
ho_nopr	2048	4096	1.29	0.00	1.28	0.00	4777023	0.00	-	-	-	1530	515.02	4	513	-	-	-	-
ho_noprxs	2048	4096	1.25	0.00	1.25	0.00	4778041	0.00	-	-	-	2043	513.76	4	-	-	-	-	-
ho_noxs	2048	4096	1.66	0.00	1.54	0.00	4783057	0.00	0.01	0	1024	1021	512.55	2	-	-	-	-	-
hybrid	2048	4096	1.66	0.00	-	-	-	-	-	-	1326	-	-	-	-	-	-	-	-
k	2048	4096	0.80	0.17	0.02	0.45	531380	0.29	0.01	0	1022	-	-	-	-	-	-	0.41	0.34
ks	2048	4096	2.50	0.02	0.09	0.39	1287910	0.01	0.01	0	80983	-	-	-	-	-	506	-	-
ks_nopr	256	512	51.14	0.04	9.93	0.49	7408803	0.04	0.00	0	0	-	-	-	-	-	2074744	-	-
ni	2048	4096	0.05	0.00	0.02	0.00	57338	0.00	0.00	0	2042	-	-	-	-	4	-	-	-
ni_nopr	2048	4096	7.74	0.00	0.01	0.00	16785355	0.00	-	-	-	-	-	-	-	2046	-	-	-
ho	4096	8192	7.44	0.00	7.42	0.00	18932846	0.00	0.03	0	1024	1025	1032.00	1	2045	-	-	-	-
ho_nopr	4096	8192	5.77	0.00	5.76	0.00	18903061	0.00	-	-	-	3066	1027.01	4	1025	-	-	-	-
ho_noprxs	4096	8192	5.78	0.00	5.77	0.00	18905103	0.00	-	-	-	4091	1025.75	4	-	-	-	-	-
ho_noxs	4096	8192	7.85	0.00	7.39	0.00	18916498	0.00	0.02	0	2048	2045	1024.52	2	-	-	-	-	-
hybrid	4096	8192	6.72	0.00	-	-	-	-	-	-	2658	-	-	-	-	-	-	-	-
k	4096	8192	2.40	0.26	0.06	0.09	1309191	0.40	0.02	0	2046	-	-	-	-	-	-	1.31	0.96
ks	4096	8192	6.45	0.03	0.22	0.46	2738795	0.01	0.02	0	173665	-	-	-	-	-	607	-	-
ks_nopr	512	1024	222.81	0.01	26.44	0.64	31730580	0.01	0.00	0	0	-	-	-	-	-	9039839	-	-
ni	4096	8192	0.11	0.00	0.03	0.00	114682	0.00	0.02	0	4090	-	-	-	-	4	-	-	-
ni_nopr	4096	8192	38.74	0.00	0.01	0.00	67125195	0.00	-	-	-	-	-	-	-	4094	-	-	-
ho	8192	16384	36.27	0.00	36.24	0.00	75152772	0.00	0.05	0	2048	2049	2056.00	1	4093	-	-	-	-
ho_nopr	8192	16384	28.02	0.00	28.00	0.00	75110509	0.00	-	-	-	6138	2051.00	4	2049	-	-	-	-
ho_noprxs	8192	16384	28.16	0.00	28.14	0.00	75114599	0.00	-	-	-	8187	2049.75	4	-	-	-	-	-
ho_noxs	8192	16384	37.98	0.00	35.64	0.00	75152956	0.00	0.05	0	4097	4093	2048.51	1	-	-	-	-	-
hybrid	8192	16384	28.14	0.00	-	-	-	-	-	-	5319	-	-	-	-	-	-	-	-
k	8192	16384	6.34	0.10	0.11	0.07	3317116	0.18	0.05	0	4094	-	-	-	-	-	-	3.59	2.47
ks	8192	16384	18.21	0.01	0.84	0.40	5754351	0.00	0.04	0	364825	-	-	-	-	-	644	-	-
ks_nopr	1024	2048	934.99	0.01	158.37	0.65	132601552	0.01	0.00	0	0	-	-	-	-	-	38185545	-	-
ni	8192	16384	0.26	0.00	0.08	0.00	229370	0.00	0.04	0	8186	-	-	-	-	4	-	-	-
ni_nopr	8192	16384	184.54	0.00	0.03	0.00	268468171	0.00	-	-	-	-	-	-	-	8190	-	-	-
ho	16384	32768	166.92	0.00	166.86	0.00	299569833	0.00	0.10	0	4095	4097	4104.00	2	8189	-	-	-	-
ho_nopr	16384	32768	126.35	0.00	126.32	0.00	299444635	0.00	-	-	-	12282	4099.00	4	4097	-	-	-	-
ho_noprxs	16384	32768	126.70	0.00	126.66	0.00	299452821	0.00	-	-	-	16379	4097.75	4	-	-	-	-	-
ho_noxs	16384	32768	178.64	0.00	164.42	0.00	299496122	0.00	0.10	0	8192	8189	4096.51	2	-	-	-	-	-
hybrid	16384	32768	121.82	0.00	-	-	-	-	-	-	10643	-	-	-	-	-	-	-	-
k	16384	32768	13.59	0.05	0.23	0.03	6633293	0.07	0.10	0	8190	-	-	-	-	-	-	7.79	5.22
ks	16384	32768	46.36	0.00	2.29	0.64	11983898	0.01	0.08	0	764442	-	-	-	-	-	683	-	-
ni	16384	32768	0.55	0.00	0.17	0.00	458746	0.00	0.09	0	16378	-	-	-	-	4	-	-	-
ni_nopr	16384	32768	798.69	0.00	0.07	0.00	1073807307	0.00	-	-	-	-	-	-	-	16382	-	-	-
ho	32768	65536	731.91	0.00	731.78	0.00	1195948260	0.00	0.20	0	8191	8193	8200.00	2	16381	-	-	-	-
ho_nopr	32768	65536	538.90	0.00	538.83	0.00	1195863014	0.00	-	-	-	24570	8195.00	4	8193	-	-	-	-
ho_noprxs	32768	65536	540.86	0.00	540.78	0.00	1195879392	0.00	-	-	-	32763	8193.75	4	-	-	-	-	-
ho_noxs	32768	65536	803.84	0.00	719.95	0.00	1195894133	0.00	0.20	0	16385	16381	8192.50	1	-	-	-	-	-
hybrid	32768	65536	520.86	0.00	-	-	-	-	-	-	21291	-	-	-	-	-	-	-	-
k	32768	65536	-	-	-	-	-	-	0.20	0	-	-	-	-	-	-	-	-	-
ks	32768	65536	110.03	0.00	3.82	0.60	25370976	0.00	0.17	0	1621037	-	-	-	-	-	753	-	-
ni	32768	65536	1.10	0.00	0.33	0.00	917498	0.00	0.17	0	32762	-	-	-	-	4	-	-	-

Table A.31: DBLCYC data

Bibliography

[1] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved Time Bounds for the Maximum Flow Problem. *SIAM J. Comput.*, 18:939–954, 1989.

[2] R. J. Anderson and J. C. Setubal. Goldberg's Algorithm for the Maximum Flow in Perspective: a Computational Study. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18. AMS, 1993.

[3] D. L. Applegate and W. J. Cook. Personal communication. 1996.

[4] T. Badics and R. Boros. Implementing a Maximum Flow Algorithm: Experiments with Dynamic Trees. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 43–64. AMS, 1993.

[5] F. Barahona. Packing Spanning Trees. *Mathematics of Operations Research*, 20(1):104–115, 1995.

[6] R. A. Botafogo. Cluster Analysis for Hypertext Systems. In *Proc. of the 16th Annual ACM SIGIR Conference of Res. and Dev. in Info. Retrieval*, pages 116–125, 1993.

[7] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array Distribution in Data-Parallel Programs. In *Languages and Compilers for Parallel Computing*, pages 76–91. Lecture Notes in Computer Science series, vol. 896, Springer-Verlag, 1996.

[8] C. S. Chekuri, A. V. Goldberg, D. R. Karger, M. S. Levine, and C. Stein. Experimental Study of Minimum Cut Algorithms. In *Proc. 8th ACM-SIAM Symposium on Discrete Algorithms*, pages 324–333, 1997.

[9] J. Cheriyan and T. Hagerup. A randomized maximum flow algorithm. In *Proc. 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 118–123, 1989.

[10] J. Cheriyan, T. Hagerup, and K. Mehlhorn. Can a maximum flow be computed in $o(nm)$ time? In M. S. Paterson, editor, *Proc. 17th ICALP*, Lecture Notes in Computer Science #443, pages 235–248, Springer-Verlag, Berlin, 1990. An extended abstract is also available as ALCOM-90-26, ESPRIT II Basic Research Actions Program Project no. 3075 (ALCOM).

[11] J. Cheriyan and S. N. Maheshwari. Analysis of Preflow Push Algorithms for Maximum Netwrok Flow. *SIAM J. Comput.*, 18:1057–1086, 1989.

[12] B. V. Cherkassky. A Fast Algorithm for Computing Maximum Flow in a Network. In A. V. Karzanov, editor, *Collected Papers, Vol. 3: Combinatorial Methods for Flow Problems*, pages 90–96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in AMS Trans., Vol. 158, pp. 23–30, 1994.

[13] B. V. Cherkassky and A. V. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. Technical Report STAN-CS-94-1523, Department of Computer Science, Stanford University, 1994.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[15] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a Large-Scale Traveling Salesman Problem. *Oper. Res.*, 2:393–410, 1954.

[16] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.

[17] U. Derigs and W. Meier. An Evaluation of Algorithmic Refinements and Proper Data-Structures for the Preflow-Push Approach for Maximum Flow. In *ASI Series on Computer and System Sciences*, volume 8, pages 209–223. NATO, 1992.

[18] E. A. Dinic, A. V. Karzanov, and M. V. Lomonosov. On the structure of a family of minimum weighted cuts in a graph. In A. A. Fridman, editor, *Studies in Discrete Optimization*, pages 290–306. Nauka Publishers, 1976.

[19] J. Edmonds. Submodular functions, matroids, and certain polyhedra. In *Calgary International Conf. on Combinatorial Structures and their Applications*, pages 69–87, New York, 1969. Gordon and Breach.

[20] J. Edmonds. Edge-disjoint branchings. In R. Rustin, editor, *Combinatorial Algorithms*, pages 91–96, New York, 1972. Algorithmics Press.

[21] P. Elias, A. Feinstein, and C. E. Shannon. Note on Maximum Flow Through a Network. *IRE Transactions on Information Theory*, IT-2:117–199, 1956.

[22] L. R. Ford, Jr. and D. R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Math.*, 8:399–404, 1956.

[23] H. N. Gabow. A Matroid Approach to Finding Edge Connectivity and Packing Arborescences. *J. Comp. and Syst. Sci.*, 50:259–273, 1995.

[24] H. N. Gabow and H. H. Westermann. Forests, Frames and Games: Algorithms for Matroid Sums and Applications. *Algorithmica*, 7(5):465–497, 1992.

[25] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[26] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Canceling Negative Cycles. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 388–397, 1988.

[27] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *J. SIAM*, 9:551–570, 1961.

[28] D. Gusfield. Very simple methods for all pairs network flow analysis. *SIAM J. Comput.*, 19:143–155, 1990.

[29] J. Hao. A Faster Algorithm for Finding the Minimum Cut of a Graph. Unpublished manuscript, 1991.

[30] J. Hao and J. B. Orlin. A Faster Algorithm for Finding the Minimum Cut of a Graph. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, 1992.

[31] J. Hao and J. B. Orlin. A Faster Algorithm for Finding the Minimum Cut in a Directed Graph. *J. Algorithms*, 17:424–446, 1994.

[32] M. R. Henzinger and D. P. Williamson. On the Number of Small Cuts in a Graph. *Information Processing Letters*, 59:41–44, 1996.

[33] D. R. Karger. Global Min-Cuts in RNC, and Other Ramifications of a Simple Min-Cut Algorithm. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, 1993.

[34] D. R. Karger. Random sampling in cut, flow, and network design problems. In *Proc. 26th Annual ACM Symposium on Theory of Computing*, pages 648–657, 1994. Submitted to *Math. of Oper. Res.*

[35] D. R. Karger. *Random Sampling in Graph Optimization Problems*. PhD thesis, Department of Computer Science, Stanford University, Stanford, CA 94305, 1994.

[36] D. R. Karger. Using randomized sparsification to approximate minimum cut. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pages 424–432, 1994.

[37] D. R. Karger. A randomized fully polynomial approximation scheme for the all terminal network reliability problem. In *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 11–17, 1995.

[38] D. R. Karger. Minimum Cuts in Near-Linear Time. In *Proc. 28th Annual ACM Symposium on Theory of Computing*, pages 56–63, 1996.

[39] D. R. Karger and C. Stein. An $\tilde{O}(n^2)$ Algorithm for Minimum Cuts. In *Proc. 25th Annual ACM Symposium on Theory of Computing*, pages 757–765, 1993.

[40] D. R. Karger and C. Stein. A new approach to the minimum cut problem. *J. Assoc. Comput. Mach.*, 43(4):601–640, July 1996.

[41] V. King, S. Rao, and R. Tarjan. A Faster Deterministic Maximum Flow Algorithm. *J. Algorithms*, 17:447–474, 1994.

[42] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison Wesley, 2nd edition, 1981.

[43] E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan, and D. B. Shmoys. *The Traveling Salesman Problem*. Wiley & Sons, 1985.

[44] T. Leong, P. Shor, and C. Stein. Implementation of a Combinatorial Multicommodity Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 387–406. AMS, 1993.

[45] M. V. Lomonosov and V. P. Poleskii. Lower bound of network reliability. *Problems of Information Transmission*, 7:118–123, 1971.

[46] D. W. Matula. A Linear Time $2 + \epsilon$ Approximation Algorithm for Edge Connectivity. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 500–504, 1993.

[47] H. Nagamochi and T. Ibaraki. Computing Edge-Connectivity in Multigraphs and Capacitated Graphs. *SIAM J. Disc. Meth.*, 5:54–66, 1992.

[48] H. Nagamochi, T. Ono, and T. Ibaraki. Implementing an Efficient Minimum Capacity Cut Algorithm. *Math. Prog.*, 67:297–324, 1994.

[49] C. S. J. A. Nash-Williams. Edge disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 36:445–450, 1961.

[50] Q. C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 19–42. AMS, 1993.

[51] M. Padberg and G. Rinaldi. An Efficient Algorithm for the Minimum Capacity Cut Problem. *Math. Prog.*, 47:19–36, 1990.

[52] S. A. Plotkin, D. Shmoys, and E. Tardos. Fast Approximation Algorithms for Fractional Packing and Covering. In *Proc. 32nd IEEE Annual Symposium on Foundations of Computer Science*, pages 495–504, 1991.

[53] A. Ramanathan and C. Colbourn. Counting Almost Minimum Cutsets with Reliability Applications. *Math. Prog.*, 39:253–261, 1987.

[54] D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. *J. Comput. System Sci.*, 26:362–391, 1983.

[55] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *J. Assoc. Comput. Mach.*, 32(3):652–686, 1985.

[56] R. E. Tarjan. Applications of Path Compression on Balanced Trees. *J. Assoc. Comput. Mach.*, 26(4):690–715, 1979.