Learning Algorithms with Applications to Robot Navigation and Protein Folding

by

Mona Singh

S.M., Computer Science
Harvard University
(1989)
A.B., Computer Science
Harvard University
(1989)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1995

Signature of Author ____________________
Department of Electrical Engineering and Computer Science
September, 1995

Certified by ____________________
Ronald L. Rivest
Professor of Computer Science
Thesis Supervisor

Certified by ____________________
Bonnie A. Berger
Assistant Professor of Mathematics
Thesis Supervisor

Accepted by ____________________
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Learning Algorithms with Applications to Robot Navigation and Protein Folding

by

Mona Singh

Submitted to the Department of Electrical Engineering and Computer Science
on September, 1995,
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

We consider three problems in machine learning:

- concept learning in the PAC model
- mobile robot environment learning
- learning-based approaches to protein structure prediction

In the PAC framework, we give an efficient algorithm for learning any function on k terms by general DNF. On the other hand, we show that in a well-studied restriction of the PAC model where the learner is not allowed to use a more expressive hypothesis (such as general DNF), learning most symmetric functions on k terms is NP-hard.

In the area of mobile robot environment learning, we introduce the problem of *piecemeal* learning an unknown environment. The robot must learn a complete map of its environment, while satisfying the constraint that periodically it has to return to its starting position (for refueling, say). For environments that can be modeled as grid graphs with rectangular obstacles, we give two piecemeal learning algorithms in which the robot traverses a linear number of edges. For more general environments that can be modeled as arbitrary undirected graphs, we give a nearly linear algorithm.

The final part of the thesis applies machine learning to the problem of protein structure prediction. Most approaches to predicting local 3D structures, or *motifs*, are tailored towards motifs that are already well-studied by biologists. We give a learning algorithm that is particularly effective in situations where large numbers of examples of the motif are not known. These are precisely the situations that pose significant difficulties for previously known methods. We have implemented our algorithm and we demonstrate its performance on the coiled coil motif.

Thesis Supervisor: Ronald L. Rivest, Professor of Computer Science
Thesis Supervisor: Bonnie A. Berger, Assistant Professor of Mathematics

Acknowledgements

I thank my advisors Ron Rivest and Bonnie Berger for their insights, ideas, guidance, and time. I am especially lucky to have had two advisors at MIT, and I have enjoyed working with both of them. Portions of this thesis are joint work with Ron and Bonnie, as well as with Margrit Betke, Avrim Blum, and Baruch Awerbuch. I am grateful to all of them for allowing me to include our joint work in this thesis.

Thanks to David Johnson for encouraging me to go to graduate school, Shafi Goldwasser for being on my thesis committee, Albert Meyer for much good advice, and the teachers at Indian Springs School for having a continuing impact on my life.

Thanks to Peter Kim, David Wilson and Ethan Wolf for help on the biology project. Thanks to the attendees of Ron's machine learning reading group for many helpful discussions.

I am grateful to the National Science Foundation, Ron Rivest, Bonnie Berger, Baruch Awerbuch, Arvind, and CBCL for funding me at various points in my graduate career.

Thanks to Scott Bloomquist, David Jones and Bruce Dale for their help around MIT. Special thanks to Be Hubbard for interesting conversation, laughter, and much help.

Thanks to all the students in the MIT theory group for providing such a wonderful environment. Special thanks to Margrit Betke for being my officemate and good friend for all these years. Thanks to my other terrific officemates: Avrim Blum, Bronwyn Eisenberg, Joe Kilian, John Leo, and especially Javed Aslam. Special thanks to Rosario Gennaro for much help during Fall 1993. Thanks also to Ruth Bergman, Lenore Cowen, Scott Decatur, Aditi Dhagat, Rainer Gawlick, Lalita Jagdeesan, Dina Kravets, Roberto Segala, Donna Slonim, Mark Smith, David Williamson, Ethan Wolf, and Yiqun Yin.

Thanks to my friends from the "outside" who have kept me sane. I especially thank Cindy Argo, Lisa Barnard, Kelly Bodnar, Anupam Chander, Lara Embry, Kyung In Han, Mike Hase, Pam Lipson, Lucy Song, Barbara Van Gorder, Robert Weaver, Tracey Drake Weber, and Irene Yu. Special thanks to Alisa Dougless, who has provided hours of entertainment while I've been at graduate school.

I especially thank Trevor Jim for being the best friend I could ever imagine.

Most of all, I thank my family, *sine qua non.* I thank my uncles, aunts and cousins for much help throughout the years. Finally, I am grateful beyond words to my parents, and to my brothers, Rajiv and Jimmy.

Publication Notes

This technical report is a revised version of the thesis I submitted in September 1995. Chapter 2 is joint work with Avrim Blum. An extended abstract describing this work appeared in *Proceedings of the Third Annual Workshop on Computational Learning Theory* [23]. The first part of Chapter 3 is joint work with Margrit Betke and Ron Rivest. This research is described in the journal *Machine Learning* [18]. An extended abstract of this work also appeared in *Proceedings of the Sixth Conference on Computational Learning Theory* [17]. The second part of this chapter is joint work with Baruch Awerbuch, Margrit Betke and Ron Rivest. An extended abstract describing this work appeared in *Proceedings of the Eighth Conference on Computational Learning Theory* [5]. Chapter 4 is joint work with Bonnie Berger, and a paper describing this work is to appear in the *First Annual Conference on Computational Molecular Biology*.

Table of Contents

Chapter 1

Introduction

There are many reasons we want machines, or computers, to learn. A machine that can learn is able to use its experience to help itself in the future. Such a machine can improve its performance on some task after performing the task several times. This is useful for computer scientists, since it means we do not have to consider all the possible scenarios a machine might encounter. Such a machine is able to adapt to various conditions or environments, or even to changing environments. A machine that is able to learn can also help push science forward. It may be able to speed up the learning process for humans, or it may be able discern patterns or do things which humans are incapable of doing. For example, we may want to build a machine that can learn patterns that aid in medical diagnosis, or that may be able to learn how to understand and process speech. Or we might want to build an autonomous robot that can learn to walk through difficult or unexpected terrain, or that can learn a map of its environment. This robot could then be used to explore environments that are too dangerous for humans, such as the surface of other planets.

In this thesis, we study three particular problems in machine learning. In order to study any machine learning problem, we must first specify the model of learning we are interested in. There are many different possible models, and a model should be chosen according to the learning application we are interested in. Once we have specified the model we are looking at, we can give algorithms and show results within the model. There are several things which any "model of learning" must specify [69, 72, 44]:

1. **Learner:** Who is doing the learning? In this thesis, we consider the learner to be a machine, such as a computer or a robot. Sometimes the machine is assumed to have limited computational power (e.g., the machine is a finite automaton), but in this thesis we assume that the machine is as powerful as a Turing machine.

2. **Domain:** What is being learned? One of the most well-studied types of learning is *concept learning* where the learner is trying to come up with a "rule" to separate positive examples from negative examples. For example, the learner may be trying to distinguish chairs from things which are not chairs. There are many other types of things that can be learned, such as an unknown environment (e.g., a new city) or an unknown technique (e.g., how to drive).

3. **Prior Knowledge:** What does the learner know about the domain initially? This generally restricts the learner's uncertainty and/or biases and expectations about unknown domains. This tells what the learner knows about what is possible or probable in the domain. For example, the learner may know that the unknown concept is representable in a certain way. That is, the unknown concept might be known to be representable as a disjunction of features, or as a graph.

4. **Information Source:** How is the learner informed about the domain? The learner may be given labeled examples. For instance, the learner may be given examples of things which are chairs, and examples of things which are not chairs. The learner may get information about a domain by asking questions of a teacher (e.g, "Is a stool a chair?"). The learner may get information about its domain by actively experimenting with it (e.g, it may learn a map of a new city by walking around in it).

5. **Performance Criteria:** How do we know whether, or how well, the learner has learned? Different performance criteria include *accuracy* and *efficiency*. For accuracy, the learner may be evaluated by its error rate, its correctness of description, or the number of mistakes it made during learning. For efficiency, the learner may be evaluated on the amount of computation it does and the amount of information it needs (e.g., the number of examples it needs). In addition, the learner may be required to have a particular hypothesis representation of an unknown concept, or it may only need to have predictive output (i.e.,

the learner does not need a representation of the unknown concept, just a way to label new instances as either positive or negative).

Different applications require different models of machine learning. In this thesis, we consider three models of machine learning. The first part of the thesis studies a theoretical model of concept learning. For this model, we study learnability and give an efficient algorithm for learning a family of concept classes. The second part of the thesis studies mobile robot navigation and environment learning. We introduce a model of exploration, which we call *piecemeal learning*, and give efficient algorithms for piecemeal learning unknown environments. The final part of the thesis applies machine learning to the problem of protein structure prediction. We introduce a learning technique that helps gather information on protein structures that biologists are interested in, but do not know much about yet.

We now give a more detailed summary of this thesis, and outline some of the contributions of this thesis to machine learning, mobile robot navigation, and protein structure prediction.

Concept learning in the PAC framework

Much of the machine learning literature has been devoted to the problem of concept learning. We study concept learning in the Probably Approximately Correct (PAC) framework [74]. The object of a PAC learning algorithm is to approximately infer an unknown concept that belongs to some known concept class. For our purposes, it suffices to view the problem as finding a concept consistent with a given set of labeled examples. Figure 1.1 shows the information presented to the learner at the start of learning, and what the learner must produce in order to learn. The examples are assumed to be a "representative sample" of future examples the learner might see. Performance is measured by the number of examples used for learning, the time-complexity of the learning algorithm, and the accuracy of the learned concept. We consider two standard versions of the PAC model: in one, the learner is required to produce as output a hypothesis belonging to the same class as the concept to be learned, and in the other, the learner's hypothesis can be any polynomial-time algorithm.

For this model, we study the problem of learning the concept classes of functions on k terms. Concept classes that can be represented by functions on k terms include k-term DNF (disjunctive normal form formulae with at most k terms), k-term exclusive-or, and r-of-k-term threshold functions. We give an efficient algorithm for PAC-learning any function on k terms by general DNF. We also show that for most symmetric functions on k terms, if the learner is required to output a hypothesis of the same concept class, then learning is NP-complete. Thus, our results illustrate the importance of hypothesis representation. In particular, for most concept classes of symmetric functions on k terms, learning the concept by itself is hard, but learning it by general DNF is easy.

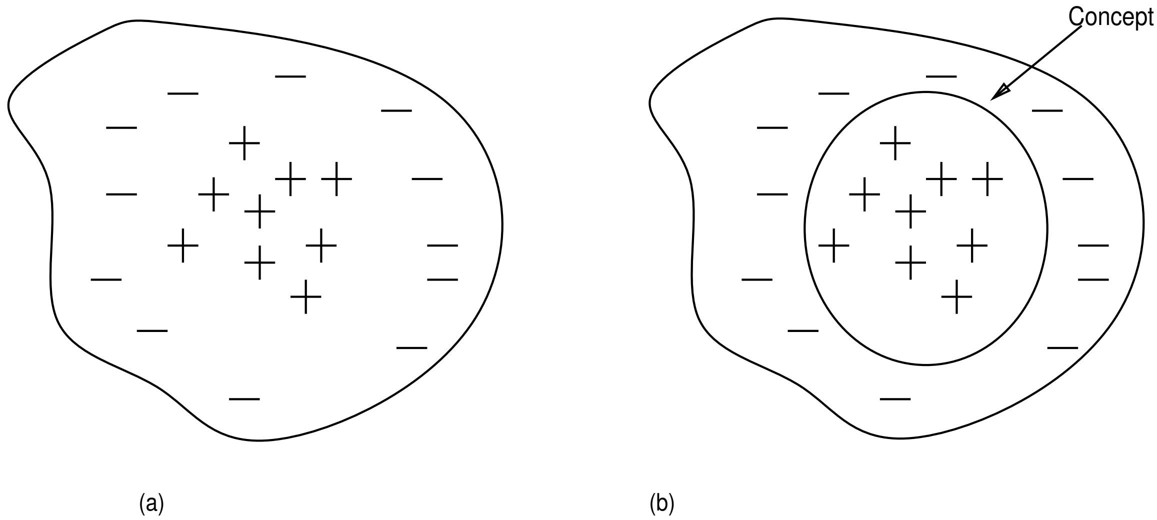
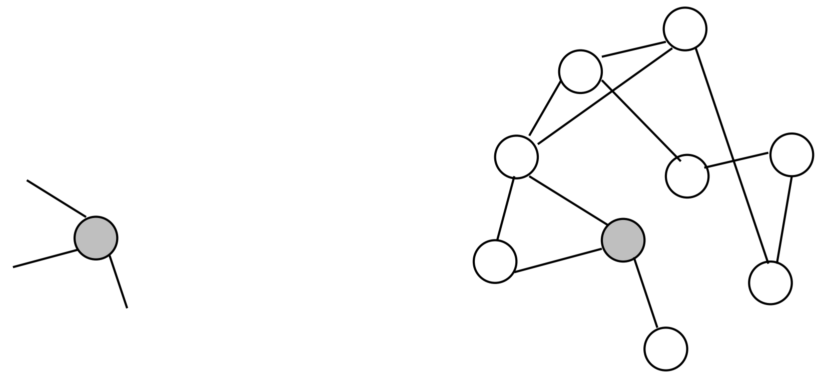


Figure 1.1: Concept learning with labeled examples. (a) Initially, the learner is given a set of labeled examples. The positive examples are denoted by $+$, and the negative examples are denoted by $-$. (b) The goal of the learner is to find a concept consistent with these examples. That is, the learner wants to find a rule that differentiates the positive examples from the negative examples.

Environment learning

In the second part of this thesis, we consider an active learning model where an autonomous robot must learn a map of its environment (see Figure 1.2). No examples are presented to the robot. Instead, it learns about the environment through active experimentation: it walks around in the environment. We introduce the problem of *piecemeal learning* of an unknown environment. The robot's goal is to learn a complete map of its environment, while satisfying the constraint that it must return every so often to its starting position. The piecemeal constraint models situations in which the robot must learn "a piece at a time." Unlike previous environment learning work, our work does not assume that the robot has sufficient resources to complete its learning task in one continuous phase; this is often an unrealistic assumption, as robots have limited power. After some exploration, the robot may need to recharge or refuel. Or, the robot may be exploring a dangerous environment, and after some time it may need to "cool down" or get maintenance. Or, the robot might have some other task to perform, and the piecemeal constraint enables "learning on the job."

The environment is modeled as an arbitrary, undirected graph, which is initially unknown to the robot. The learner's performance is measured by the number of edges it traverses while exploring. For environments that can be modeled as grid graphs with rectangular obstacles, we give two piecemeal learning algorithms in which the robot explores every vertex and edge in the graph by traversing a linear number of edges. For more general environments that can be modeled by an undirected graph, we give a piecemeal learning algorithm in which the robot traverses at most a nearly linear number of edges.



(a) (b)

Figure 1.2: Environment learning. (a) Initially the learner only knows its starting location. (b) The learner must build a map of its environment.

Learning-based methods for protein structure prediction

In the last part of this thesis, we again turn to concept learning, but here the learner is given both labeled and unlabeled examples (see Figure 1.3). Unlike the previous concept learning model, here the labeled examples that the learner is given are not representative of the examples that the learner will see; moreover, the learner knows that this is the case. Unlike the other work in this thesis, the performance measure we use here is empirical and not theoretical. Within this model, we look at the particular application of protein structure prediction.

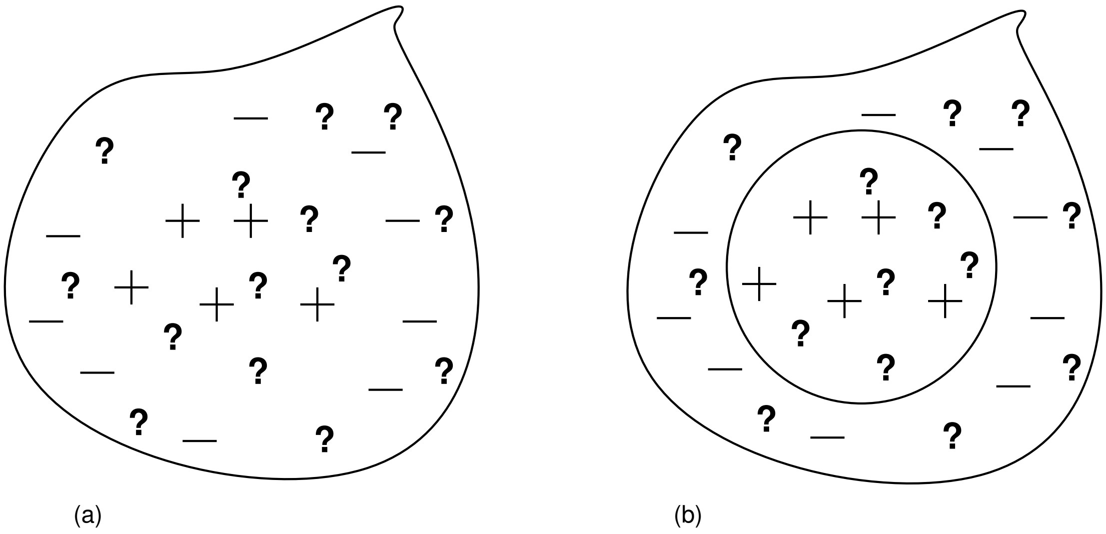


Figure 1.3: Concept learning with labeled and unlabeled examples. (a) The learner is given a set of labeled examples as well as a set of unlabeled examples. The positive examples are denoted by $+$, the negative examples are denoted by $-$, and the unlabelled examples are denoted by **?**. (b) The learner must find a concept which partitions these examples. The unlabeled points within the circle are assumed positive, and the unlabeled points outside of the circle are assumed negative.

The goal of this work is to use computational techniques to learn about protein structures or folds which biologists do not yet know much about. Current techniques for predicting local three-dimensional structures, or *motifs*, are tailored towards folds which are already well-studied and documented by biologists. We give a learning algorithm that is particularly effective in situations where this is not the case. We generalize the 2-stranded coiled coil domain to learn 3-stranded coiled coils, and perhaps other similar motifs. As a consequence of this work, we have identified many new sequences that we believe contain coiled coil and coiled-coil-like structures. These sequences contain regions that are not identified by the best previous computational method, but are identified by our method. These sequences include mouse hepatitis virus, human rotavirus, human T-cell lymphotropic virus, Human Immunodeficiency Virus (HIV) and Simian Immunodeficiency Virus (SIV). Independently, recent laboratory work has predicted the existence of a coiled-coil-like structure in HIV and SIV [19, 56], and our algorithm is able to predict the regions of this structure to within a few residues. We hope that biologists will direct their laboratory efforts towards testing other new candidate sequences which we identify.

Organization of thesis

The thesis is organized in three self-contained chapters. In Chapter 2, we study the problem of learning concept classes of functions on k terms in the PAC framework. In Chapter 3, we introduce the problem of piecemeal learning unknown environments, and give efficient algorithms for this problem. In Chapter 4, we study the problem of learning protein motifs. Finally, in Chapter 5, we finish with some concluding remarks.

Chapter 2

Learning functions on k terms

2.1 Introduction

Since its introduction, Valiant's distribution-free or PAC learning framework [74] has been a well-studied model of concept learning. In this framework, the object of a learning algorithm is to approximately infer an unknown target concept that belongs to some known concept class. The learner is given examples chosen randomly according to a fixed but unknown distribution. The goal of the learner is to find (with high probability) a hypothesis that accurately predicts new instances as positive or negative examples of the concept. We consider here two standard versions of this model: in one, the learner is required to produce as output a hypothesis belonging to the same class as the target concept, and in the other, the learner's hypotheses may be any polynomial-time algorithm [64][50][66]. Several examples are known of concept classes that are hard to learn when hypotheses are restricted to belong to the same class as the target concept but easy to learn when they may belong to a larger class. In particular, Pitt and Valiant [64] showed that learning the class of k-term DNF formulas (that is, functions that can be represented by a disjunction of k monomials) is NP-hard if the learner is required to produce a k-term DNF formula, but is easy if the learner may use a representation of k-CNF formulas.

In this chapter, we show that this phenomenon occurs for a broad class of formulas. In particular, given constant k and function f, let $\mathcal{C}_{k,f}$ be the class of concepts of the form $f(T_1, \ldots, T_k)$ where $T_1, \ldots, T_k$ are monomials. So, for example, if f is the OR function then $\mathcal{C}_{k,f}$ is the class of

k-term DNF formulas. We show that for any symmetric function f (that is, f depends on only the number of inputs which are 1), learning the class $\mathcal{C}_{k,f}$ by hypothesis class $\mathcal{C}_{k,f}$ is NP-hard except for $f \in \{\wedge, \neg\wedge, T, F\}$. The hardness result completely characterizes the complexity of learning $\mathcal{C}_{k,f}$ by $\mathcal{C}_{k,f}$ for symmetric functions f. For $f \in \{T, F\}$, learning $\mathcal{C}_{k,f}$ is trivial, and for $f \in \{\wedge, \neg\wedge\}$, $\mathcal{C}_{k,f}$ is the class of conjunctions or disjunctions respectively, so learning $\mathcal{C}_{k,f}$ by $\mathcal{C}_{k,f}$ is easy by a standard procedure for learning monomials.

On the other hand, we also present a polynomial-time algorithm that learns the class of $\mathcal{C}_k$ of *all* concepts $f(T_1, \ldots, T_k)$, where f is any $\{0, 1\}$-valued function of k inputs and $T_1, \ldots, T_k$ are monomials, using a hypothesis class of general DNF. As a consequence, this algorithm will learn by DNF the concept classes $\mathcal{C}_{k,f}$ for which learning $\mathcal{C}_{k,f}$ by $\mathcal{C}_{k,f}$ is NP-hard.

A strategy for learning the special case of k-term DNF formulas is to learn by the hypothesis class of k-CNF (that is, conjunctions of disjunctions of size k). Every k-term DNF can be written as a k-CNF (since we can "distribute out" the k-term DNF) and k-CNF can be easily learned by standard procedures. Suppose, however, that we wish to learn in the same manner another class of concepts $\mathcal{C}_{k,f}$ (that is, other than k-term DNF) for which learning $\mathcal{C}_{k,f}$ by $\mathcal{C}_{k,f}$ is NP-hard. Our results and related results by Fischer and Simon [41] show that exclusive-or (XOR) is one such function. In this case, an XOR of k monomials need not be representable as a k-CNF or as a k-DNF (for example, $x_1x_2 \oplus x_3$ written as a DNF requires one term of size 3, and written as a CNF requires one clause of size 3). In addition an XOR of k monomials need not have representation as a conjunction of XORs of size k. Thus, the standard strategy for learning k-term DNF or k-term CNF will not work for learning k-term XOR.

Instead, our algorithm is based on a different strategy. Roughly, we use the fact that a monomial can be made false just by setting one of the literals that appears in it to 0. So, given a concept represented by a function on k unknown terms $T_1, \ldots, T_k$, if we are able to "guess" literals that appear in $k - 1$ of the monomials and consider only examples in which these monomials are false, we can then focus on the term remaining. Then, once we have been able to classify the examples that satisfy only one term of $T_1, \ldots, T_k$, we can focus on those that satisfy pairs of terms, and so on.

2.2 Notation and definitions

We will consider learning over the Boolean domain $X_n = \{0,1\}^n$. An *example* is an element $\vec{v} \in \{0,1\}^n$ and a *concept* c is a boolean function on examples. A *concept class* is a collection of concepts. For a given a target concept c, a *labeled example* for c is a pair $\langle \vec{v}, c(\vec{v}) \rangle$ where $\vec{v}$ is a *positive example* if $c(\vec{v}) = 1$ and a *negative example* if $c(\vec{v}) = 0$. For convenience, we will at times think of an example as a collection of variables or attributes x. In this case, for an example $\vec{v}$ and variable $x \in X_n$, let $\vec{v}(x) = 1$ if the bit of $\vec{v}$ corresponding to x is 1, and 0 otherwise. Also, we will use $|c|$ to denote the size of concept c under some reasonable encoding.

Let k be a constant. Define the concept class $\mathcal{C}_k$ to be the set of all concepts $f(T_1, \ldots, T_k)$ where $T_1, \ldots, T_k$ are monomials (conjunctions of literals) and f is any $\{0,1\}$-function on k boolean inputs. For example, class $\mathcal{C}_2$ includes the concept $x_1\overline{x}_2 \oplus x_3x_4x_5$, where "$\oplus$" denotes the XOR function. For a given function f, let $\mathcal{C}_{k,f}$ be those concepts in $\mathcal{C}_k$ of the form $f(T_1, \ldots, T_k)$ for the given f. We say that a function f is *symmetric* if the value of f depends only on the number of inputs that are 1. For a symmetric function f and integer i, we let $f(i)$ denote the value of f when exactly i of its inputs are 1.

We study learning in the distribution-free or Probably Approximately Correct (PAC) learning model [74, 2]. In the PAC learning model, we assume that the learning algorithm has available an oracle EXAMPLES(c) that when queried, produces a labeled example $\langle \vec{v}, c(\vec{v}) \rangle$ according to a fixed but unknown probability distribution D. If C and H are concept classes, we say that algorithm A *learns* C *by* H if for some polynomial p, for all target concepts $c \in C$, distributions D, and error parameters ε and δ: algorithm A halts in time $p(n, \frac{1}{\varepsilon}, \frac{1}{\delta}, |c|)$ and outputs a hypothesis $h \in H$ that with probability at least $1 - \delta$ has error at most ε. The error of a hypothesis h is the probability that $h(\vec{v}) \neq c(\vec{v})$ when $\vec{v}$ is chosen from the distribution D.

For the purposes of our positive results, it will be enough to consider the following sufficient condition for learnability [26]. An algorithm A is an "Occam algorithm" for C if on any sample (collection of labeled examples) of size m consistent with some $c \in C$, algorithm A produces a consistent hypothesis of size at most $|c|^\beta m^\alpha$ for constants $\alpha < 1, \beta \geq 1$. Blumer et al. show that any Occam algorithm for C, producing hypotheses from H, will learn C by H.

2.3 The learning algorithm

In this section, we present an algorithm that learns the class $\mathcal{C}_k$ by the hypothesis class of general DNF. To illustrate the strategy used, let us consider first the problem of learning an XOR of two monotone monomials.

Suppose the target concept is $c = T_1 \oplus T_2$ for monotone monomials T_1 and T_2. We know each positive example $\vec{v}$ satisfies one of T_1 or T_2 and fails to satisfy the other, and so has some $v_i = 0$ for x_i in exactly one of T_1 and T_2. Given a set S of examples, let S_i, for $1 \leq i \leq n$, be the set of those examples $\vec{v}$ for which $v_i = 0$. If a variable x_i is contained in exactly one of $\{T_1, T_2\}$, say x_i is in T_1, then the monomial $\overline{x}_i \wedge T_2$ is satisfied by every positive example in S_i and no negative example in S. Therefore, we can actually *find* a monomial consistent with the positive examples in this S_i and the negative examples in S, using the standard monomial learning procedure.

So, we can learn an XOR of two terms as follows. For each variable x_i, find a monomial M_i consistent with positive examples in S_i and with all negative examples, if such a monomial exists. Then, output as hypothesis the disjunction of the M_i's. The hypothesis produced is consistent with every negative example since no negative example satisfies any M_i. Also, since every positive example lies in some S_i for x_i in exactly one of $\{T_1, T_2\}$, for each positive example we will have found some monomial it satisfies.

We now present an Occam algorithm based on the above strategy that learns the class $\mathcal{C}_k$ using a hypothesis class of DNF. Without loss of generality, we may assume that the target concept is some $f(T_1, \ldots, T_k)$ where the T_i are monotone (we can think of non-monotone terms as monotone terms over the attribute space $\{x_1, \overline{x}_1, x_2, \overline{x}_2, \ldots, x_n, \overline{x}_n\}$). The algorithm Learn-$k$-Term takes as input a set S of m examples consistent with some function $f(T_1, \ldots, T_k)$ on k monotone monomials and outputs a DNF of size $O(n^{k+1})$ consistent with the given examples.

The basic idea of Learn-k-Term is as follows. In the first iteration, the algorithm "handles" those positive examples that satisfy none of the terms. That is, if there are any such positive examples, the algorithm finds a set of monomials such that each of those positive examples satisfies one of the monomials. These monomials are then added to the DNF being built. In the second iteration, the algorithm tries to find a set of monomials for those positive examples that satisfy exactly one of the terms. This process is continued so that at each iteration the algorithm focuses on examples that satisfy an increasing number of terms. Thus, at each value of r in the loop, the algorithm finds terms to handle all the positive examples that do not satisfy exactly r terms of the target concept. The ordering of $r = k$ down to 0 is important to ensure that needed terms are not thrown away in step 9. Note that in step 5, we allow the i_j to be the same. This is done for purposes of simpler analysis—the algorithm would still work if we just considered the $\binom{n}{r}$ sets of r different variables.

Learn-k-Term(S)
1 Let $\mathbf{P}$ = the positive examples in S
2 Let $\mathbf{N}$ = the negative examples in S
3 Initialize the DNF hypothesis h to $\{\}$.
4 **For** $r = k$ down to 0 **Do**
5   **For** each set of r variables: $\{x_{i_1}, \ldots, x_{i_r}\}$ **Do**
6   Let M be the monomial $\overline{x}_{i_1} \cdots \overline{x}_{i_r}$.
7   Let U be the set of those examples $\vec{v} = (v_1, \ldots, v_n) \in \mathbf{P}$ such that $v_{i_1} = v_{i_2} = \ldots = v_{i_r} = 0$. That is, U is the set of examples in $\mathbf{P}$ satisfying the term M.
8   Let T be the monomial that is the conjunction of all x_i such that every example $\vec{v} \in U$ has $v_i = 1$. (T is the most specific monotone monomial satisfied by all examples in U.)
9   **If** no negative example in $\mathbf{N}$ satisfies term $MT = \overline{x}_{i_1}\overline{x}_{i_2} \cdots \overline{x}_{i_r}T$
10   **Then**
11   add MT as a term to the hypothesis h
12   let $\mathbf{P} \leftarrow \mathbf{P} - U$.

Algorithm Learn-k-Term clearly runs in time polynomial in m and n^k, so we just need to prove the following theorem.

Theorem 1 *Algorithm* Learn-k-Term*, on m examples consistent with some function f of k monotone monomials over $\{0,1\}^n$, produces a consistent DNF hypothesis of size $O(n^{k+1})$.*

Proof: First notice the following facts. The DNF h produced by algorithm Learn-k-Term has at most $n^k + n^{k-1} + \ldots + n = O(n^k)$ terms of size $O(n)$, so the size of the hypothesis is at most $O(n^{k+1})$. Also, the hypothesis h is consistent with the set $\mathbf{N}$ of negative examples, since in step 9 any term that some negative example satisfies will never be included in the DNF. Thus all we need to do is prove that for every positive example $\vec{v} \in \mathbf{P}$, there is some term added to h which is satisfied by $\vec{v}$.

Let $f(T_1, \ldots, T_k)$ be the target concept where $T_1, \ldots, T_k$ are monotone monomials. Let S_j for $j \in \{0, \ldots, k\}$ be the set of those positive examples seen that satisfy exactly j of $T_1, \ldots, T_k$ (if f is the XOR function, for instance, then the sets S_j for even values of j are all empty). We will argue by induction on the index j; in particular we will argue that after the iteration of the loop of Learn-k-Term in which $r = k - j$, all positive examples $\vec{v} \in S_j$ have been "captured" by (that is, they satisfy) some term in h.

$\mathbf{j = 0, r = k}$: Let $\vec{v}$ be a positive example that satisfies none of $T_1, \ldots, T_k$. If such an example exists, then any other example satisfying none of $T_1, \ldots, T_k$ must also be a positive example. There must be some collection of variables $x_{i_1} \in T_1, \ldots, x_{i_k} \in T_k$ (not necessarily all different) such that $v_{i_1} = v_{i_2} = \ldots = v_{i_k} = 0$, or otherwise $\vec{v}$ would satisfy some term.

Consider the iteration in which the monomial M is $\overline{x}_{i_1} \cdots \overline{x}_{i_k}$. Example $\vec{v}$ satisfies M and so is put into U in step 7. Any other example satisfying M cannot satisfy any of $T_1, \ldots, T_k$ (by definition of $x_{i_1}, \ldots, x_{i_k}$) and therefore must be positive. So, a term MT satisfied by $\vec{v}$ will be added to h in step 4.

$\mathbf{j > 0, r = k - j}$: Let $\vec{v}$ be a positive example that satisfies exactly j of the terms $T_1, \ldots, T_k$; for convenience, assume $\vec{v}$ satisfies terms $T_{r+1}, \ldots, T_k$. Any other example satisfying exactly those terms and no others must also be positive. Let $x_{i_1} \in T_1, \ldots, x_{i_r} \in T_r$ be a collection of not necessarily distinct variables such that $v_{i_1} = \ldots = v_{i_r} = 0$.

At the iteration in which the monomial M is $\overline{x}_{i_1} \cdots \overline{x}_{i_r}$, example $\vec{v}$ is put into set U in step 7 and the term T created is satisfied by $\vec{v}$. In fact, T also has in it all variables contained in the terms $T_{r+1}, \ldots, T_k$. The reason is as follows:

> Suppose x_i is contained in one of $T_{r+1}, \ldots, T_k$ but not in T. Then, there must exist some positive example $\vec{w} \in U$ such that $w_i = 0$. So, example $\vec{w}$ fails to satisfy at least one of $T_{r+1}, \ldots, T_k$ in addition to not satisfying any of $T_1, \ldots, T_r$. But, this means that $\vec{w}$ satisfies fewer than j terms and so must already have been removed from $\mathbf{P}$ in an earlier iteration by our inductive hypothesis. (Note

that it is for this reason that algorithm Learn-k-Term begins with $r = k$ and works down to $r = 0$.)

So, any example satisfying MT must satisfy all of $T_{r+1}, \ldots, T_k$ (since it satisfies T) and none of $T_1, \ldots, T_r$ (since it satisfies M) and therefore must be positive. Thus, term MT will be added to h in step 9.

So, we have shown that algorithm Learn-k-Term, on any size input consistent with some function f of k monotone monomials over $\{0,1\}^n$, produces a consistent hypothesis of size $O(n^{k+1})$ in time polynomial in m and n^k. ∎

Corollary 1 *The concept class $\mathcal{C}_k$ is learnable by DNF in the distribution-free model.*

In fact, if we assume without loss of generality that the target concept $c = f(T_1, \ldots, T_k)$ has the property that $f(00\cdots0) = 0$ (otherwise we will learn $\overline{c}$), then we can start algorithm Learn-k-Term at $r = k-1$ and produce a DNF of only $O(n^{k-1})$ terms instead of one of $O(n^k)$ terms. So, for example, we can learn a k-term DNF with a DNF hypothesis of $O(n^{k-1})$ terms each of size $O(n)$. This differs from the standard procedure of learning k-term DNF, which gives a k-CNF of $O(n^k)$ clauses of size $k = O(1)$. Moreover, if we know that f outputs 0 when only a few of its inputs are 1, then we can produce a hypothesis of smaller size. For example, if f is the majority function, then we can start Learn-k-Term with $r = k/2$ and get a DNF of only $O(n^{k/2})$ terms.

2.3.1 Decision lists

An alternative way to learn $\mathcal{C}_k$ is to learn by the class of k-decision lists (k-DLs).[1] In fact, the proof for Algorithm Learn-**k**-Term can be modified to show any concept in $\mathcal{C}_k$ can be written as a k-decision list. In particular, let $c = f(T_1, \ldots, T_k)$ be some concept in $\mathcal{C}_k$. The decision list will consist of rules of the form "if M_i then b_i," where the each M_i will correspond to one of the monomials M in algorithm LEARN-k-TERM.

[1] A *k-decision list* is a function of the form: "`if` M_1 `then` b_1`, else if` M_2 `then` b_2`, else` ... `else if` M_m `then` b_m `else` b_{m+1}," where the M_i are monomials of size at most k and the b_i are each either 0 or 1.

Let b_0 be the value of $c(x)$ when x satisfies none of $T_1, \ldots, T_k$. Put on the top of the decision list all rules of the from "if $\overline{x}_{i_1}\overline{x}_{i_2}\cdots\overline{x}_{i_k}$ then b_0," where $x_{i_1} \in T_1, \ldots, x_{i_k} \in T_k$. Let us say that a set of rules "captures" an example if the example satisfies the if-portion of one of them. Thus, we have now captured all examples that satisfy none of the T_i (and have classified them correctly).

Inductively suppose we have created rules that capture (and correctly classify) all examples satisfying $j-1$ or fewer of the k terms. Append onto the bottom of the decision list the following rules. For each subset $\{T_{t_1}, \ldots, T_{t_{k-j}}\} \subseteq \{T_1, \ldots, T_k\}$ such that all examples which satisfy exactly the j terms remaining are positive, add all rules of the form: "if $\overline{x}_{i_1}\overline{x}_{i_2}\cdots\overline{x}_{i_{k-j}}$ then 1," where $x_{i_1} \in T_{t_1}, \ldots, x_{i_{k-j}} \in T_{t_{k-j}}$. For each subset $\{T_{t_1}, \ldots, T_{t_{k-j}}\} \subseteq \{T_1, \ldots, T_k\}$ such that all examples satisfying exactly the j terms remaining are negative, add all rules of the form: "if $\overline{x}_{i_1}\overline{x}_{i_2}\cdots\overline{x}_{i_{k-j}}$ then 0," where $x_{i_1} \in T_{t_1}, \ldots, x_{i_{k-j}} \in T_{t_{k-j}}$.

Finally, the default case of the decision list is the rule "else b," where b is the classification of examples satisfying all the terms T_i. It is clear from the above arguments that this k-decision list is logically equivalent to the k-term function.

The mistake-bound model is a model of learning more stringent than the PAC model; here, unlabeled examples are presented to the learner in an arbitrary order, and after each one the learner must predict its classification before being told the correct value. The learner is judged by the total number of mistakes it makes in such a sequence. Using the halving algorithm [54], k-decision lists can be learned in the mistake-bound model with $O(n^k)$ mistakes. Thus have the following theorem:

Theorem 2 *All functions on k terms can be learned in the mistake-bound model with $O(n^k)$ mistakes, using a representation of k-decision lists.*

In fact, we can learn k-term functions in an "attribute-efficient" sense, where the number of mistakes is polynomial in the number of relevant variables (variables that appear in some term T_i) and is only logarithmic in the number of irrelevant variables. This uses a result of Littlestone [54] as follows.

An *alternation* in a decision list is a pair of adjacent rules such that the boolean classification values for the rules differ. By appropriately ordering the rules in the decision list construction above (listing the "negative rules" before the "positive rules" on alternate j values) one can see that for any k-term function there is a logically equivalent k-decision list with at most k alternations. Such a decision list can be thought of as a function in the form:

> if ($M_{1,1}$ OR $M_{1,2}$ OR ... OR M_{1,m_1}) then b_1, else if ($M_{2,1}$ OR $M_{2,2}$ OR ... OR M_{2,m_2}) then b_2, else ... else if ($M_{k-1,1}$ OR $M_{k-1,2}$ OR ... OR $M_{k-1,m_{k-1}}$) then b_{k-1} else b_k,

where $b_i = 1 - b_{i-1}$.

Decision lists with small numbers of alternations can be written as linear threshold functions over the monomials M_i, with not too large integral weights. For instance, if $b_{k-1} = 1$, k is odd, and m is the sum of the m_i, the above decision list can be written as:

$$\begin{aligned}(M_{k-1,1} + \ldots + M_{k-1,m_{k-1}}) \quad &- \quad m(M_{k-2,1} + \ldots + M_{k-2,m_{k-2}}) \\ &+ \quad m^2(M_{k-3,1} + \ldots + M_{k-3,m_{k-3}}) \\ &\vdots \\ &- \quad m^k(M_{1,1} + \ldots + M_{1,m_1}) \qquad > \quad 0.\end{aligned}$$

If only r variables are relevant to the k-term function, then the number of rules m is at most r^k. Therefore, the maximum weight in the threshold function is r^{k^2}.

Littlestone [54] gives an algorithm that can be used to learn such a function, where the number of mistakes is at most $O((mr^{k^2})^2 \log(n^k)) = O(kr^{2k+2k^2} \log n)$. Thus, if the number r of relevant variables is small, this can be a savings in the number of mistakes made. Thus we have the following theorem:

Theorem 3 *Any function on k terms can be learned with $O(kr^{2k+2k^2} \log n)$ mistakes, where r is the number of relevant variables.*

2.4 Hardness results

In this section, we show that learning the class $\mathcal{C}_{k,f}$ often requires allowing the learning algorithm a more expressive hypothesis class than $\mathcal{C}_{k,f}$. In the previous section, we gave an algorithm that learns the concept class of functions on k terms using the hypothesis class of general DNF. On the other hand, we now show that when learning the class $\mathcal{C}_{k,f}$, if the algorithm must produce a hypothesis from the class $\mathcal{C}_{k,f}$, the problem can become NP-hard. In particular, we show that for any symmetric function f, learning the class $\mathcal{C}_{k,f}$ by hypothesis class $\mathcal{C}_{k,f}$ is NP-hard except for $f \in \{\wedge, \neg\wedge, T, F\}$. The hardness result completely characterizes the complexity of learning $\mathcal{C}_{k,f}$ by $\mathcal{C}_{k,f}$ for symmetric functions f. For $f \in \{T, F\}$, learning $\mathcal{C}_{k,f}$ is trivial, and for $f \in \{\wedge, \neg\wedge\}$, $\mathcal{C}_{k,f}$ is the class of conjunctions or disjunctions respectively, so learning $\mathcal{C}_{k,f}$ by $\mathcal{C}_{k,f}$ is easy by a standard procedure. We show the following:

Theorem 4 *For any symmetric function f on k inputs except for $f \in \{\wedge, \neg\wedge, T, F\}$, learning the class $\mathcal{C}_{k,f}$ by $\mathcal{C}_{k,f}$ is NP-hard.*

This theorem extends the work of Pitt and Valiant [64], which shows that learning the class of k-term DNF formulas is NP-hard if the learner is required to produce a k-term DNF formula. Before giving the proof of Theorem 4, we first provide some intuition. For $k \geq 3$, the proof of Pitt and Valiant is essentially a reduction from graph k-colorability.[2] Their reduction is as follows. Given the graph, they create a variable x_i for each vertex $v_i \in V$. They then create one positive examples for each vertex so that the example corresponding to vertex i has bit i set to 0 and all other bits set to 1. They also create one negative example for each edge such that the example corresponding to edge (i, j) has bits i and j set to 0 and the other bits set to 1. They then show that the set of examples is consistent with a disjunction of k terms if and only if G is k-colorable. Their proof does not work for more general symmetric functions f of k terms. In particular, when f is a symmetric function other than OR (e.g., when the concept class is 4-term exclusive-or formulas), using their reduction it is possible to find a formula $f(T_1, T_2, \ldots, T_k)$ that correctly classifies all positive and negative examples, but the corresponding coloring is invalid. The basic problem is that unlike the case of disjunction, for arbitrary f, as the number of inputs that are 1 increases, the value of f can switch back and forth between 1 and 0. To solve this problem, we introduce enough variables and examples for each edge such that x_i and x_j are forced to occur in different terms. We can use this

[2] The graph k-colorability problem is: given a graph $G = (V, E)$ and a positive integer k, does there exist a function $f : V \to \{1, 2, \ldots, k\}$ such that $f(u) \neq f(v)$ whenever $(u, v) \in E$? That is, using at most k colors, is it possible to assign a color to each vertex in the graph such that for any edge, its vertices are given different colors?

technique to reduce graph k-colorablity to learning any symmetric function on k terms (except $\wedge, \neg\wedge, T, F$).

To show Theorem 4, we first consider the concept class $\mathcal{C}_{k,f}^{mon} = \{f(T_1, \ldots, T_k)\}$ where $T_1, \ldots, T_k$ are *monotone* monomials, and show that learning $\mathcal{C}_{k,f}^{mon}$ by $\mathcal{C}_{k,f}^{mon}$ is NP-hard. We then give an extension of the argument that shows that learning $\mathcal{C}_{k,f}^{mon}$ by $\mathcal{C}_{k,f}$ is NP-hard. This implies Theorem 4.

Theorem 5 *For any symmetric function f on k inputs except $f \in \{\wedge, \neg\wedge, T, F\}$, learning the class $\mathcal{C}_{k,f}^{mon}$ by $\mathcal{C}_{k,f}^{mon}$ is NP-hard.*

Proof: First note that if $k = 2$ then the only functions f with $f \notin \{\wedge, \neg\wedge, T, F\}$ are the functions $\{\vee, \neg\vee, \oplus, \neg\oplus\}$. The proof of [64] for 2-term DNF can be applied directly for these cases; so, we assume that $k \geq 3$. Without loss of generality, we assume that $f(k-1) = 0$; that is, f outputs 0 when exactly $k-1$ of its inputs are 1. Otherwise, we show that learning $\mathcal{C}_{k,f'}^{mon}$ by $\mathcal{C}_{k,f'}^{mon}$ for $f' = \overline{f}$ is NP-hard and the result follows.

The proof is a reduction from graph k-colorability. Given a graph $G = (V, E)$, we create labeled examples over $n = |V| + (k-2)|E|$ variables such that there exists $c \in \mathcal{C}_{k,f}^{mon}$ consistent with these examples if and only if there is a k-coloring of the graph. We assume that G contains no isolated vertices since such vertices do not affect the coloring of the graph.

We denote the n variables as follows. There is one variable x_i for each vertex $i \in V$, and $k-2$ variables $w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{k-2}$ for each edge $(i,j) \in E$. Thus, for each edge $(i,j) \in E$, we have a set $W_{i,j}$ of k associated variables $\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{k-2}\}$. We add the $w_{i,j}$'s so that ultimately any hypothesis consistent with the examples we define must contain x_i and x_j in different terms if $(i,j) \in E$. For convenience, we use the following notation to denote an example that consists of 1's in all bits except those specified by a set of variables W.

- For W a collection of variables, let $g(W)$ be the example $\vec{v}$ such that $\vec{v}(x) = 0$ for $x \in W$ and $\vec{v}(x) = 1$ for $x \notin W$. Recall that $\vec{v}(x)$ is the bit of $\vec{v}$ corresponding to variable x.

For $l \in \{1, \ldots, k\}$ and $(i,j) \in E$, let $S_{i,j}^l = \{g(W) : W \subseteq W_{i,j}, |W| = l\}$. That is, set $S_{i,j}^l$ is the set of examples $\vec{v} = g(W)$ for W a subset of size l of the set $\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{k-2}\}$. We now define k sets of examples as follows:

$$S^1 = \{S^1_{i,j} : (i,j) \in E\},$$

$$S^2 = \{S^2_{i,j} : (i,j) \in E\},$$

$$\vdots$$

$$S^k = \{S^k_{i,j} : (i,j) \in E\},$$

such that $\vec{v} \in S^l, 1 \leq l \leq k$, is a positive example if and only if $f(k-l) = 1$. That is, for each edge $(i,j) \in E$, each S^l contains $\binom{k}{l}$ examples corresponding to that edge. Each $\vec{v} \in S^l$ has exactly l bits set to 0, where the l variables corresponding to these bits are chosen from some set $W_{i,j}$. If f is true when exactly $k-l$ terms are true (i.e. $f(k-l) = 1$), then we label all vectors is S^l as positive examples; otherwise we label them as negative examples. For example, if f is the XOR function and k is even, then all examples in $S^1, S^3, \ldots$ are labeled as positive and those in $S^2, S^4, \ldots$ are labeled as negative.

We now show that there exist monotone terms $T_1, T_2, \ldots, T_k$ such that $f(T_1, T_2, \ldots, T_k)$ is consistent with these examples if and only if there is a k-coloring of the graph G.

($\Leftarrow$) Given a k-coloring of the graph, then for each vertex i which is colored l, place x_i in term T_l. Then for each edge (i,j), variables x_i and x_j appear in different terms. Now arbitrarily place the remaining $k-2$ variables associated with this edge (the $w_{i,j}$'s) into the remaining $k-2$ terms such that each term receives exactly one variable. Thus for each edge (i,j), each of the associated variables $\{x_i, x_j, w^1_{i,j}, w^2_{i,j}, \ldots, w^{k-2}_{i,j}\}$ occurs in a different term. So for any example in S^l, exactly l terms are false and $k-l$ terms are true. Since the examples in S^l are positive exactly when $f(k-l) = 1$, the concept $f(T_1, T_2, \ldots, T_k)$ classifies all examples correctly.

($\Rightarrow$) Suppose we have $T_1, T_2, \ldots, T_k$ such that concept $c = f(T_1, T_2, \ldots, T_k)$ is consistent with all the examples. Now color the vertices by the function $\chi : V \rightarrow \{1, 2, \ldots, k\}$ defined by $\chi(i) = \min\ \{j$: variable x_i occurs in term $T_j\}$. Lemma 1 guarantees we have a well defined function, and Lemma 2 gives us a valid coloring. ∎

Lemma 1 *Each variable x_i occurs in some term.*

Proof: Suppose that some x_i does not occur in any term. Let $q = \min\ \{l : f(k-l) = 1 \text{ and } l > 0\}$. That is, q is the smallest positive number of terms that can be false such that concept c is true. Note that q is the least index such that $c(\vec{v}) = 1$ for $\vec{v} \in S^q$. We know that q exists for $f \notin \{\text{AND, FALSE}\}$.

Pick j such that $(i,j) \in E$ (since we assumed that the graph is connected, we know some such j exists). Now consider the positive example $\vec{v} = g(\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\})$. If x_i does not occur in any term, then $\vec{u} = g(\{x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\})$ satisfies the same number of terms as $\vec{v}$, and thus $c(\vec{u}) = c(\vec{v}) = 1$. But $\vec{u}$ belongs to S^{q-1}, and we know all examples in S^{q-1} are negative examples by our definition of q (S^q is our first set of positive examples). Contradiction. ∎

Lemma 2 *If $(i,j) \in E$ then x_i and x_j never occur in the same term.*

Proof: Suppose that for $(i,j) \in E$, variables x_i and x_j occur in the same term. Again, let us look at vectors in S^q where $q = \min\ \{l : f(k-l) = 1 \text{ and } l > 0\}$. In particular, consider the positive example $\vec{v} = g(\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\})$. By Lemma 3, we know that exactly q terms of c are not satisfied by $\vec{v}$. Then we know that each of these q terms must contain at least one variable of $\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\}$. If x_i and x_j occur in the same term, then we know that some variable $x \in \{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\}$ occurs in at least two terms. Let r be the number of terms that variable x appears in. We build a set S of at most $q - r + 1$ variables such that $\vec{u} = g(S)$ also makes q terms false. Initially let $S = \{x\}$. Then for each of the remaining $q - r$ terms not satisfied by $\vec{v}$, place into S some variable from $\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\}$ which appears in that term. Now consider example $\vec{u}$. The terms not satisfied by $\vec{u}$ are exactly those not satisfied by $\vec{v}$, so $c(\vec{u}) = c(\vec{v}) = 1$. Moreover, since $S \subset \{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\} \subset W_{i,j}$, example $\vec{u}$ must lie in some set S^l where $l < q$. But S^q is our first set (the set of least index) of positive examples, so $\vec{u}$ must be negative. Contradiction. ∎

Lemma 3 *Exactly q terms of c are not satisfied by $\vec{v}$.*

Proof: Suppose not. That is, suppose $r \neq q$ terms of c are not satisfied by $\vec{v}$. Since $\vec{v}$ is a positive example, $f(k-r) = 1$ and by definition of q we have $r > q$. There are now two cases:

Case 1: $f(k-l)=1$ for all $l \in \{q, q+1, \ldots, r\}$.

By definition of q, for any set $S \subset \{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\}$ of size $q-1$, $c(g(S))=0$. This implies that each $\vec{u}=g(S)$ satisfies at least $r-q+1$ more terms of $\{T_1, \ldots, T_k\}$ than does $\vec{v}$. But this requires each variable in $\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\}$ to appear without any other variable from this set in $r-q+1$ terms. So there must exist $q(r-q+1)$ terms not satisfied by $\vec{v}$. Since $r>q$ and $q \neq 1$ (we know $f(k-q)=1$ but $f(k-1)=0$), we have:

$$\begin{aligned} r(q-1) &> q(q-1) \\ rq-r &> q^2-q \\ q(r-q+1) &> r. \end{aligned}$$

Thus, more than r terms are not satisfied by $\vec{v}$. Contradiction.

Case 2: $f(k-l)=0$ for some $l \in \{q+1, \ldots, r-1\}$.

Consider the sequence of examples:

$$\begin{aligned} \vec{v}_q &= g(\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-2}\}), \\ \vec{v}_{q+1} &= g(\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{q-1}\}), \\ &\vdots \\ \vec{v}_k &= g(\{x_i, x_j, w_{i,j}^1, w_{i,j}^2, \ldots, w_{i,j}^{k-2}\}). \end{aligned}$$

We assign values to q_i, r_i, and l_i which maintain the following invariants: $q_i < l_i < r_i$ and $f(k-q_i)=f(k-r_i)$ and $f(k-q_i) \neq f(k-l_i)$. Initially let $q_1=q$, $r_1=r$, and $l_1=l$. Initially, positive example $\vec{v}_{q_1}$ fails to satisfy r_1 terms and there exists l_1 between q_1 and r_1 with $f(k-l_1)=0$. Thus negative example $\vec{v}_{l_1}$ must fail to satisfy some $r_2>r_1$ terms. Now let $q_2=l_1$ and $l_2=r_1$, and so we have $f(k-q_2)=f(k-r_2)=0$, $f(k-l_2)=1$, and $q_2<l_2<r_2$. Thus we know that positive example $\vec{v}_{l_2}$ must satisfy some $r_3>r_2$ terms. Letting $q_3=l_2$ and $l_3=r_2$, and continuing in this fashion, we find an increasing sequence $q_1, q_2, q_3, \ldots$, such that each example $\vec{v}_{q_i}$ fails to satisfy $r_i>q_i$ terms. At $q_i=k$, we have a contradiction. ∎

We have now finished proving Theorem 5. We now extend the proof to the general case in which the terms $T_1, \ldots, T_k$ may be non-monotone.

Proof of Theorem 4: We show that $\mathcal{C}_{k,f}^{mon}$ by $\mathcal{C}_{k,f}$ is NP-hard. This implies the theorem. Given a graph $G = (V, E)$ we create a new graph G' consisting of $k+1$ copies $G_1, \ldots, G_{k+1}$ of G. Clearly G' is k-colorable if and only if G is. We define examples in the same way as in the proof of Theorem 5. We must now show that there exist (non-monotone) terms $T_1, T_2, \ldots, T_k$ such that $f(T_1, T_2, \ldots, T_k)$ is consistent with the examples if and only if there is a k coloring of the graph G. Given a k-coloring of the graph G, we can easily find a k-coloring of graph G'. From this coloring, we can find k terms such that $f(T_1, T_2, \ldots, T_k)$ is consistent with the examples, using the same method as in the proof of Theorem 5. For the other direction, we must show that if there are non-monotone terms $T_1, \ldots, T_k$ such that $f(T_1, \ldots, T_k)$ is consistent with the examples, then G is k-colorable. Notice that if any term T_l has in it a negated variable corresponding to a vertex or edge of some graph G_q, then T_l is not satisfied by any example corresponding to graph G_r for $r \neq q$. If term T_l has in it negated variables from more than one graph G_q, then no examples satisfy term T_l, and thus the concept is equivalent to the concept with term T_l replaced by 0. If T_l contains negated variables corresponding to a vertex or edge of just one graph G_q, then we can replace term T_l by 0 and mark graph G_q; this new concept is still consistent with the examples corresponding to all unmarked graph copies. We continue this procedure until all terms left have no negated variables. We never mark all the graph copies since we mark at most one graph for each term that is set to 0, and there are more graphs than terms. So, since each term left has no negated variables we can color any one of the remaining unmarked graphs using the coloring given in the proof of Theorem 5. ∎

2.5 Conclusion

We present an algorithm that learns the class $\mathcal{C}_k$ of all concepts $f(T_1, \ldots, T_k)$ where f is a $\{0,1\}$-valued function and $T_1, \ldots, T_k$ are monomials, using a hypothesis class of general DNF. We also show that learning the class $\mathcal{C}_{k,f}$ by $\mathcal{C}_{k,f}$ where f is a symmetric function is NP-hard, except for $f \in \{\wedge, \neg\wedge, T, F\}$ for which learning is easy. We leave as open the problem of classifying the learnability of $\mathcal{C}_{k,f}$ by $\mathcal{C}_{k,f}$ for more general functions f.

Chapter 3

Piecemeal learning of unknown environments

3.1 Introduction

We address the situation where a robot, to perform a task better, must learn a complete map of its environment. The robot's goal is to learn this map while satisfying the *piecemeal constraint* that learning must be done "a piece at a time." Why might mobile robot exploration be done piecemeal? Robots may have limited power, and after some exploration they may need to recharge or refuel. In addition, robots may explore environments that are too risky or costly for humans to explore, such as the inside of a volcano (e.g., CMU's Dante II robot), or a chemical waste site, or the surface of Mars. In these cases, the robot's hardware may be too expensive or fragile to stay long in dangerous conditions. Thus, it may be best to organize the learning into phases, allowing the robot to return to a start position for refueling and maintenance.

The "piecemeal constraint" means that each of the robot's exploration phases must be of limited duration. We assume that each exploration phase starts and ends at a fixed start position. This special location might be a refueling station or a base camp. Between exploration phases the robot might perform other unspecified tasks. Piecemeal learning thus enables "learning on the job", since the phases of piecemeal learning can help the robot improve its performance on the other tasks it performs. This is the "exploration/exploitation tradeoff": spending some time exploring (learning) and some time exploiting what one has learned.

The piecemeal constraint can make efficient exploration surprisingly difficult. We first consider piecemeal learning in environments that can be modeled as grid graphs with rectangular obstacles. For these environments, we give two linear-time algorithms. The first algorithm, the "wavefront" algorithm, can be viewed as an optimization of breadth-first search for our problem. The second algorithm, the "ray" algorithm, can be viewed as a variation on depth-first search. We then extend these results by giving a nearly linear algorithm for piecemeal learning more complicated environments that can be modeled by arbitrary undirected graphs. For piecemeal learning of these environments, we give some "approximate" breadth-first search algorithms. We first give a simple algorithm that runs in $O(E + V^{1.5})$ time. We then improve this algorithm and give a nearly linear time algorithm: it achieves $O(E+V^{1+o(1)})$ running time. An interesting open problem is whether arbitrary, undirected graphs can be learned piecemeal in linear time.

We now give a brief summary of the rest of this chapter. Section 3.2 gives some related work on environment learning and mobile robot navigation. Section 3.3 formalizes our model. Section 3.4 discusses piecemeal learning of arbitrary graphs, and the problems with some initial approaches. Section 3.5 gives an approximate solution to the off-line version of this problem. In addition, it gives our strategy for solving the problem we are interested in (the on-line version of the problem). Section 3.6 introduces the notion of "city-block" graphs, discusses shortest paths in such graphs, and gives two linear time algorithms for piecemeal learning these types of graphs. Section 3.7 considers piecemeal learning of general graphs, and gives a nearly linear algorithm for this problem. Section 3.8 gives an application of our algorithms to the problem of finding a treasure in an unknown, potentially infinite graph. Finally, Section 3.9 concludes with some open problems.

3.2 Related work

Theoretical approaches to environment learning differ in how the robot's environment is modeled, what types of sensors the robot has, the accuracy of the robot's sensor, if the robot has access to a teacher, and what the performance measure is. The robot's environment is often modeled by a finite automaton, a directed graph, an undirected graph, or some special case of the above. Typically, it is assumed that the robot knows what type of environment it is trying to learn. The robot may have vision, or may have no long-range sensors whatsoever. Sometimes the robot is assumed to have accurate sensors, and in other models the robot's sensors may be noisy. Performance measures for the robot's accuracy vary from requiring the robot to always output an exact map of the environment, to requiring that the robot output a good map with high probability. Performance in terms of efficiency can be judged by either the total number of steps taken by the robot, the number of queries the robot may have to ask of a teacher, competitive ratios (e.g., the total number of steps the robot makes divided by the minimum number of steps required had the robot known the environment), or some other measure.

Rivest and Schapire [70] study environments that can be modeled by a strongly connected deterministic finite automata. The robot gets information about the automaton by actively experimenting in the environment and by observing input-output behavior. Rivest and Schapire show that a robot with a teacher can with high probability learn such an environment. They use homing sequences to improve Angluin's algorithm [1] to learn without using a "reset" mechanism. Ron and Rubinfeld [71] further extend this result by giving an efficient algorithm that with high probability learns finite automata with small cover time, without requiring a teacher. Dean et al. [33] study the problem of learning finite automaton when the output at each state has some probability of being incorrect. They give an algorithm for learning finite automata, assuming that the robot has access to a distinguishing sequence. Freund et al. [43] give algorithms for learning "typical" deterministic finite automata from random walks.

Deng and Papadimitriou [35] and Betke [16] model the robot's environment as a directed graph, with distinct and recognizable vertices and edges. They give a learning algorithm with a constant competitive ratio when the graph is Eulerian or when the deficiency of the graph is 1. For general graphs, they give a competitive ratio that is exponential in the deficiency of the graph. Bender and Slonim [11] look at the more complicated case of directed graphs with indistinguishable vertices. They show that a single robot with a constant number of pebbles cannot learn such environments without knowing the size of the graph. On the other hand, they give a probabilistic algorithm for two cooperating robots to learn such an environment. Dudek et al. [38] study the easier problem of learning undirected graphs with indistinguishable vertices, and give an algorithm for a robot with one or markers to learn such an environment.

Deng, Kameda, and Papadimitriou [34] model environments such as "rooms" as polygons with polygonal obstacles. They assume the robot has vision, and must learn a map of the room. They show that if the polygon has an arbitrary number of polygonal obstacles in it, then then it is not possible to achieve a constant competitive ratio. For the simplified case of a rectilinear room with no obstacles, they show a $2\sqrt{2}$ competitive algorithm for learning the room. Kleinberg [52] improves this to a $\frac{5}{4}\sqrt{2}$ competitive algorithm. For a rectilinear room with at most k obstacles, Deng et al. give an algorithm with $O(k)$ competitive ratio. They also give constant competitive algorithms for environments that are modeled by general polygons with a bounded number of obstacles, but the constant they give is large.

There has also been much theoretical work in the case where the robot's goal is to get from one point to another in an unknown environment. The robot learns parts of the environment as it is navigating, but its primary goal is to reach a particular location. In some cases, the robot knows exactly where there the goal location is, and in others it is assumed that the robot will recognize the goal location.

Baeza-Yates, Culberson and Rawlins [8] study the *cow path problem.* The robot must search for an object in an unknown location on 2 or more rays (the endpoints of the rays are at some fixed start position). They give an optimal deterministic strategy for this problem. For the case of 2 rays, they use a doubling strategy and get a competitive ratio of 9; they extend this technique for m rays and get a competitive ratio of $1 + 2(m^m/(m-1)^{m-1})$. Kao, Reif and Tate [49] give a randomized algorithm for this problem that has better expected performance than any deterministic algorithm. Kao, Ma, Sipser and Yin [48] give an optimal deterministic search strategy for the case of multiple robots.

Papadimitriou and Yanakakis [62] consider the problem of a robot with vision moving around in a plane filled with obstacles. The robot does not know its environment, but knows its exact absolute location at all times, as well as its start position and its goal position. The robot's goal is to travel from the start position to the goal position. Papadimitriou and Yanakakis show that for the case of non-touching axis parallel rectangular obstacles, the competitive ratio is $\Omega(\sqrt{n})$, where n is the length of the shortest path between the start and goal locations. For the case of square obstacles, they give a $\frac{1}{3}\sqrt{26} \approx 1.7$ competitive algorithm, and show that any strategy must have competitive ratio greater than $\frac{3}{2}$.

Blum, Raghavan, and Schieber [22] also study the problem of point to point navigation in an unknown two-dimensional geometric environment with convex obstacles. For the case of axis parallel rectangular obstacles, they give an algorithm with competitive ratio $O(\sqrt{n})$, matching the lower bound of Papadimitriou and Yanakakis. They also introduce and give an algorithm for the *room problem*, where the goal of the robot is to go from a point on a wall of the room to a specified point in the center of the room. The room contains axis parallel obstacles, but the obstacles do not touch the sides of the wall. Bar-Eli, Berman, Fiat, and Yan [10] show that any algorithm for this problem has competitive ratio $\Omega(\log n)$, and give an algorithm attaining this bound.

Blum and Chalasani [21] consider the point to point problem in an unknown environment when the robot makes repeated trips between two points. The goal of the robot is to find better paths in each trip. In environments with axis parallel obstacles, they give an algorithm with the property that at the i-th trip, the robot's path is $O(\sqrt{n/i})$ times the shortest path length.

Klein [51] considers the problem of a polygon with distinguished start and goal vertices. The robot's goal is to walk inside the polygon from the start location to the goal location. The goal location is recognized as soon as the robot sees it. For a special type of polygon known as a *street*, Klein gives an algorithm with a $1 + \frac{3}{2}\pi \approx 5.71$ competitive ratio. Kleinberg [52] improves this by giving an algorithm with competitive ratio $\sqrt{4+\sqrt{8}} \approx 2.61$. For rectilinear streets, the algorithm achieves a competitive ratio of $\sqrt{2}$.

There are many other related papers in the literature, particularly in the area of robotics (e.g., [57]) and maze searching (e.g., [25, 24]). Rao, Kareti, Shi, and Iyengar [68] give a survey of work on robot navigation in unknown terrains.

3.3 Formal model

We model the robot's environment as a finite connected undirected graph $G = (V, E)$ with distinguished start vertex s. Vertices represent accessible locations. Edges represent accessibility: if $\{x, y\} \in E$ then the robot can move from x to y, or back, in a single step.

We assume that the robot can always recognize a previously visited vertex; it never confuses distinct locations. At any vertex the robot can sense only the edges incident to it; it has no vision or long-range sensors. The robot can distinguish between incident edges at any vertex. Each edge has a label that distinguishes it from any other edge. Without loss of generality, we can assume that the edges are ordered. At a vertex, the robot knows which edges it has traversed already. The robot only incurs a cost for traversing edges; thinking (computation) is free. We also assume a uniform cost for an edge traversal. We consider the running time of a piecemeal learning algorithm to be the number of edge traversals made by the robot.

The robot is given an upper bound B on the number of steps it can make (edges it can traverse) in one exploration phase. In order to assure that the robot can reach any vertex in the graph, do some exploration, and then get back to the start vertex, we assume B allows for at least one round trip between s and any other single vertex in G, and also allows for some number of exploration steps. More precisely, we assume $B = (2 + \alpha)r$, where $\alpha > 0$ is some constant, and r is the *radius* of the graph (the maximum of all shortest-path distances between s and any vertex in G).

Initially all the robot knows is its starting vertex s, the bound B, and the radius r of the graph. The robot's goal is to explore the entire graph: to visit every vertex and traverse every edge, minimizing the total number of edges traversed.

3.4 Initial approaches to piecemeal learning

A simple approach to piecemeal learning of arbitrary undirected graphs is to use an ordinary search algorithm—breadth-first search (BFS) or depth-first search (DFS)—and just interrupt the search as needed to return to visit s. (Detailed descriptions of BFS and DFS can be found in algorithms textbooks [32].) Once the robot has returned to s, it goes back to the vertex at which search was interrupted and resumes exploration. We now illustrate the problems each of these approaches has for efficient piecemeal learning.

Depth-first search

In depth-first search, edges are explored out of the most recently discovered vertex v that still has unexplored edges leaving it. When all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discovered. This process continues until all edges are explored. This search strategy, without interruptions due to the piecemeal constraint, is efficient since at most $2|E|$ edges are traversed. Interruptions, or exploration in phases of limited duration, complicate matters. For example, suppose in the first phase of exploration, at step $B/2$ of a phase the robot reaches a vertex v as illustrated in Figure 3.1. Moreover, suppose that the only path the robot knows from s to v has length $B/2$. At this point, the robot must stop exploration and go back to the start location s. In the second phase, in order for the robot to resume a depth-first search, it should go back to v, the most recently discovered vertex. However, since the robot only knows a path of $B/2$ to v, it cannot proceed with exploration from that point.

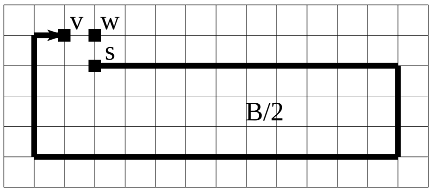


Figure 3.1: The robot reaches vertex v after $B/2$ steps in a depth-first search. Then it must interrupt its search and return to s. It cannot resume exploration at v to get to vertex w, because the known return path is longer than $B/2$, the remaining number of steps allowed in this exploration phase. DFS fails.

Since DFS with interruptions fails to reach all the vertices in the graph, another approach to solve the piecemeal learning problem would be to try a *bounded depth-first search* strategy. In bounded DFS, edges are explored out of the most recently discovered vertex v which had depth less than a given bound β. However, a straightforward bounded DFS strategy also does not translate into an efficient piecemeal learning algorithm for arbitrary undirected graphs.

Breadth-first search

Unlike depth-first search, breadth-first search with interruptions does guarantee that all vertices in the graph are ultimately explored. Whereas a DFS strategy cannot resume exploration at vertices to which it only knows a long path, a BFS strategy can always resume exploration. This is because BFS ensures that the robot always knows a *shortest* path from s to any explored vertex. However, since a BFS strategy explores all the vertices at the same distance from s before exploring any vertices that are further away from s, the resulting algorithm may not be efficient. Note that in the usual BFS model, the algorithm uses a queue to keep track of which vertex it will search from next. Thus, searching requires extracting a vertex from this queue.

In our model, however, since the robot can only search from its current location, extracting a vertex from this queue results in a *relocation* from the robot's current location to the location of the new vertex. Unlike the standard BFS model, our model does not allow the robot to "teleport" from one vertex to another; instead, we consider a *teleport-free* exploration model, where the robot must physically move from one vertex to the next.

In BFS, the robot may not move further away from the source than the unvisited vertex nearest to the source. At any given time in the algorithm, let Δ denote the shortest-path distance from s to the vertex the robot is visiting, and let δ denote the shortest-path distance from s to the vertex nearest to s that is as yet unvisited. With traditional breadth-first search we have $\Delta \leq \delta$ at all times. With teleport-free exploration, it is generally impossible to maintain $\Delta \leq \delta$ without a great loss of efficiency:

Lemma 4 *A robot which maintains $\Delta \leq \delta$ (such as a traditional BFS) may traverse $\Omega(E^2)$ edges.*

Proof: Consider the graph in Figure 3.2, where the vertices are $\{-n, -n+1, \ldots, -1, s = 0, 1, 2, \ldots, n-1, n\}$, and edges connect consecutive integers. To achieve $\Delta \leq \delta$, a teleport-free BFS algorithm would run in quadratic time, traveling back and forth from 1 to -1 to -2 to 2 to 3 .... ∎

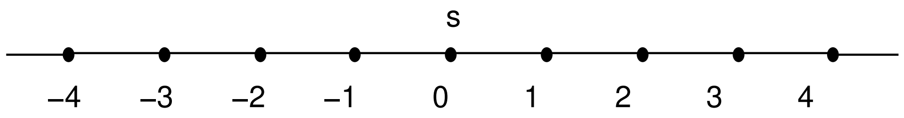


Figure 3.2: A simple graph for which the cost of BFS is quadratic in the number of edges.

3.5 Our approaches to piecemeal learning

In this section, we discuss our approach to piecemeal learning of general graphs. First we define the off-line version of this problem, and give an approximate solution for it, and then we give a general method for converting certain types of search algorithms into piecemeal learning algorithms.

3.5.1 Off-line piecemeal learning

We now develop a strategy for the off-line piecemeal learning problem which we can adapt to get a strategy for the on-line piecemeal learning problem.

In the *off-line* piecemeal learning problem, the robot is given a finite connected undirected graph $G = (V, E)$, a start location $s \in V$, and a bound B on the number of edges traversed in any exploration phase. The robot's goal is to plan an optimal search of the graph that visits every vertex and traverses every edge, and also satisfies the piecemeal constraint (i.e., each exploration phase traverses at most B edges and starts and ends at the start location). Note that since the graph is given, the problem does not actually have a learning or exploration component. However, for simplicity we continue using "learning" and "exploration."

The off-line piecemeal learning problem is similar to the well-known *Chinese Postman Problem* [39], but where the postman must return to the post-office every so often. (We could call the off-line problem the *Weak Postman Problem*, for postmen who cannot carry much mail.) The same problem arises when many postmen must cover the same city with their routes.

The Chinese Postman Problem can be solved by a polynomial time algorithm if the graph is either undirected or directed [39]. The Chinese Postman problem for a mixed graph that has undirected and directed edges was shown to be NP-complete by Papadimitriou [61]. We do not know an optimal off-line algorithm for the Weak Postman Problem; this may be an NP-hard problem.

We now give an approximation algorithm for the off-line piecemeal learning problem using a simple "interrupted-DFS" approach.

Theorem 6 *There exists an approximate solution to the off-line piecemeal learning problem for an arbitrary undirected graph $G = (V, E)$ which traverses $O(|E|)$ edges.*

Proof: Assume that the radius of the graph is r and that the number of edges the robot is allowed to traverse in each phase of exploration is $B = (2 + \alpha)r$, for some constant α such that αr is a positive integer. Before the robot starts traversing any edges in the graph, it looks at the graph to be explored, and computes a depth-first search tree of the graph. A depth-first traversal of this depth-first search tree defines a path of length $2|E|$ which starts and ends at s and which goes through every vertex and edge in the graph. The robot breaks this path into segments of length αr. The robot also computes (off-line) a shortest path from s to the start of each segment.

The robot then starts the piecemeal learning of the graph. Each phase of the exploration consists of taking a shortest path from s to the start of a segment, traversing the edges in the segment, and taking a shortest path back to the start vertex. For each segment, the robot traverses at most $2r$ edges to get to and from the segment, and αr edges to explore the segment itself. Thus, since the total number of edge traversals for each segment is at most $(2+\alpha)r = B$, the piecemeal constraint is satisfied. Since there are $\lceil \frac{2|E|}{\alpha r} \rceil$ segments, there are $\lceil \frac{2|E|}{\alpha r} \rceil - 1$ interruptions, and the number of edge traversals due to interruptions is at most:

$$\begin{aligned}\left(\left\lceil \frac{2|E|}{\alpha r} \right\rceil - 1\right) 2r &\leq \left(\frac{2|E|}{\alpha r}\right) 2r \\ &\leq \frac{4|E|}{\alpha}\end{aligned}$$

Thus the total number of edge traversals is at most $(4/\alpha + 2)|E| = O(E)$. ∎

3.5.2 On-line piecemeal learning

We now show how we can change the strategy outlined above to obtain an efficient on-line piecemeal learning algorithm.

We call an on-line search *optimally interruptible* if it always knows a shortest path back to s that can be composed from the edges that have been explored. We refer to a search as *efficiently interruptible* if it always knows a path back to s via explored edges of length at most the radius of the graph.

Theorem 7 *An efficiently interruptible algorithm for exploring an unknown graph $G = (V, E)$ with n vertices and m edges that takes time $T(n, m)$ can be transformed into a piecemeal learning algorithm that takes time $O(T(n, m))$.*

Proof: The proof of this theorem is similar to the proof of Theorem 6. However, there are a few differences. Instead of using an ordinary search algorithm (like DFS) and interrupting as needed to return to s, we use an efficiently interruptible search algorithm. Moreover, the search is on-line and is being interrupted during exploration. Finally, the cost of the search is not $2|E|$ as in DFS, but at most $T(n,m)$.

Assume that the radius of the graph is r and that the number of edges the robot is allowed to traverse in each phase of exploration is $B = (2+\alpha)r$, for some constant α such that αr is a positive integer. In each exploration phase, the robot will execute αr steps of the original search algorithm. At the beginning of each phase the robot goes to the appropriate vertex to resume exploration. Then the robot traverses αr edges as determined by the original search algorithm, and finally the robot returns to s. Since the search algorithm is efficiently interruptible, the robot knows a path of distance at most r from s to any vertex in the graph. Thus the robot traverses at most $2r + \alpha r = B$ edges during any exploration phase.

Since there are $\lceil\frac{T(n,m)}{\alpha r}\rceil$ segments, there are $\lceil\frac{T(n,m)}{\alpha r}\rceil - 1$ interruptions, and the number of edge traversals due to interruptions is:

$$\begin{aligned}\left(\left\lceil\frac{T(n,m)}{\alpha r}\right\rceil - 1\right) 2r &\leq \frac{T(n,m)}{\alpha r} 2r \\ &\leq \frac{2T(n,m)}{\alpha}\end{aligned}$$

Thus, the total number of edge traversals is $T(n,m) + 2T(n,m)/\alpha = T(n,m)(1+2/\alpha) = O(T(n,m))$. ■

For arbitrary undirected planar graphs, we can show that any optimally interruptible search algorithm requires $\Omega(|E|^2)$ edge traversals in the worst case. For example, exploring the graph in Figure 3.2 (known initially only to be an arbitrary undirected planar graph) would result in $|E|^2$ edge traversals if the search is required to be optimally interruptible.

Because it seems difficult to handle arbitrary undirected graphs efficiently, we first focus our attention on a special class of undirected planar graphs. These graphs, known as *city-block* graphs, are defined in the Section 3.6.1. For these graphs we present two efficient $O(|E|)$ optimally interruptible search algorithms. Since an optimally interruptible search algorithm is also an efficiently interruptible search algorithm, these two algorithms give efficient piecemeal learning algorithms for city-block graphs. The wavefront algorithm is a modification of breadth-first search that is optimized for city-block graphs. The ray algorithm is a variation on depth-first search. For piecemeal learning arbitrary undirected graphs, since optimally interruptible search algorithms are not efficient, we look at efficiently interruptible search algorithms. In particular, our algorithms are *approximate* breadth-first search algorithms.

3.6 Linear time algorithms for city-block graphs

This section first defines and motivates the class of city-block graphs, and then develops some useful properties of such graphs that will be used in Subsections 3.6.2 (which gives the wavefront algorithm for piecemeal learning of a city-block graph) and 3.6.3 (which gives the ray algorithm).

Both the wavefront algorithm and the ray algorithm are optimally interruptible, and thus maintain at all times knowledge of a shortest path back to s. Since BFS is optimally interruptible, we study BFS in some detail to understand the characteristics of shortest paths in city-block graphs. Our algorithms depend on the special properties that shortest paths have in city-block graphs. We also study BFS because our wavefront algorithm is a modification of BFS.

3.6.1 City-block graphs

We model environments such as cities or office buildings in which efficient on-line robot navigation may be needed. We focus on grid graphs containing some non-touching axis-parallel rectangular "obstacles". We call these graphs *city-block graphs*. They are rectangular planar graphs in which all edges are either vertical (north-south) or horizontal (east-west), and in which all faces (city blocks) are axis-parallel rectangles whose opposing sides have the same number of edges. A 1×1 face might correspond to a standard city-block; larger faces might correspond to obstacles (parks or shopping malls). Figure 3.3 gives an example. City-block graphs are also studied by Papadimitriou and Yanakakis [62], Blum, Raghavan, and Schieber [22], and Bar-Eli, Berman, Fiat and Yan [10].

An $m \times n$ city-block graph with no obstacles has exactly mn vertices (at points (i, j) for $1 \leq i \leq m$, $1 \leq j \leq n$) and $2mn - (m + n)$ edges (between points at distance 1 from each other). Obstacles, if present, decrease the number of accessible locations (vertices) and edges in the city-block graph. In city-block graphs the vertices and edges are deleted such that all remaining faces are rectangles.

We assume that the directions of incident edges are apparent to the robot.

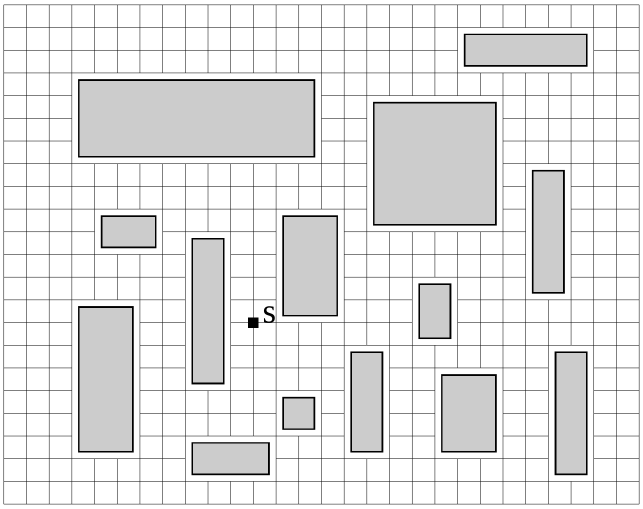


Figure 3.3: A city-block graph with distinguished start vertex s.

Let $\delta(v, v')$ denote the length of the shortest path between v and v', and let $d[v]$ denote $\delta(v, s)$, the length of the shortest path from v back to s.

Monotone paths and the four-way decomposition

A city-block graph can be usefully divided into four regions (north, south, east, and west) by four monotone paths: an east-north path, an east-south path, a west-north path, and a west-south path. The east-north path starts from s, proceeds east until it hits an obstacle, then proceeds north until it hits an obstacle, then turns and proceeds east again, and so on. The other paths are similar (see Figure 3.4). Note that all monotone paths are shortest paths. Furthermore, note that s is included in all four regions, and that each of the four monotone paths (east-north, east-south, west-north, west-south) is part of all regions to which it is adjacent.

In Lemma 5 we show that for any vertex, there is a shortest path to s through only one region. Without loss of generality, we therefore only consider optimally interruptible search algorithms that divide the graph into these four regions, and search these regions separately. We only discuss what happens in the northern region; the other regions are handled similarly.

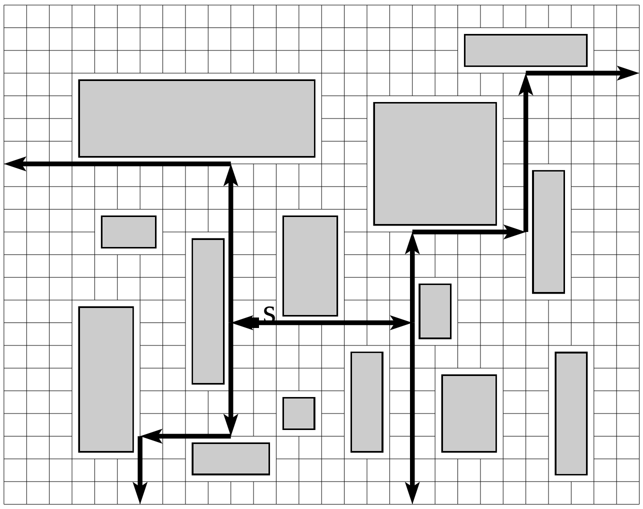


Figure 3.4: The four monotone paths and the four regions.

Lemma 5 *There exists a shortest path from s to any point in a region that only goes through that region.*

Proof: Consider a point v in some region A. Let p be any shortest path from s to the point v. If p is not entirely contained in region A, we can construct another path p' that is entirely contained in region A. We note that the vertices and edges which make up the monotone paths surrounding a region A are considered to be part of that region.

Since path p starts and ends in region A but is not entirely contained in region A, there must be a point u that is on p and also on one of the monotone paths bordering A. Note that u may be the same as v. Without loss of generality, let u be the last such point, so that the portion of the path from u to v is contained entirely within region A. Then the path p' will consist of the shortest path from s to u along the monotone path that u is on, followed by the portion of p from u to v. This path p' is a shortest path from s to v because p was a shortest path and p' can be no longer than p. ∎

Canonical shortest paths of city-block graphs

We now make a fundamental observation on the nature of shortest paths from a vertex v back to s. In this section, we consider shortest paths in the northern region; properties of shortest paths in other region are similar.

Lemma 6 *For any vertex v in the northern region, there is a canonical shortest path from v to the start vertex s which goes south whenever possible. The canonical shortest path goes east or west only when it is prevented from going south by an obstacle or by the monotone path defining the northern region.*

Proof: We call the length $d[v]$ of the shortest path from v to s the *depth* of vertex v. We show this lemma by induction on the depth of a vertex.

For the base case, it is easy to verify that any vertex v such that $d[v] = 1$ has a canonical shortest path that goes south whenever possible.

For the inductive hypothesis, we assume that the lemma is true for all vertices that have depth $t-1$, and we want to show it is true for all vertices that have depth t. Consider a vertex p at depth t. If there is an obstacle obstructing the vertex that is south of point p or if p is on a horizontal segment of the monotone path defining the northern region, then it is impossible for the canonical shortest path to go south, and the claim holds. Thus, assume the point south of p is not obstructed by an obstacle or by the monotone path defining the northern region. Then we have the following cases:

Case 1: Vertex p_s directly south of p has depth $t-1$. In this case, there is clearly a canonical shortest path from p to s which goes south from p to p_s and then follows the canonical shortest path of p_s, which we know exists by the inductive assumption.

Case 2: Vertex p_s directly south of p has depth not equal to $t-1$. Then one of the remaining adjacent vertices must have depth $t-1$ (otherwise it is impossible for p to have depth t). Furthermore, none of these vertices has depth less than $t-1$, for otherwise vertex p would have depth less than t.

Note that the point directly north of p cannot have depth $t-1$. If it did, then by the inductive hypothesis, it has a canonical shortest path which goes south. But then p has depth $t-2$, which is a contradiction.

Thus, either the point west of p or the point east of p has depth $t-1$. Without loss of generality, assume that the point p_w west of p has depth $t-1$. We consider two subcases. In case (a), there is a path of length 2 from p_w to p_s that goes south one step from p_w, and then goes east to p_s. In case (b), there is no such path.

Case (a): If there is such a path, the vertex directly south of p_w exists, and by the inductive hypothesis has depth $t-2$ (since there is a canonical shortest path from p_w to s of length $t-1$, the vertex directly to the south of p_w has depth $t-2$). Then p_s, which is directly east of this point, has depth at most $t-1$ and thus there is a canonical path from p to s which goes south whenever possible.

Case (b): Note that the only way there does not exist a path of length 2 from p_w to p_s (other than the obvious one through p) is if p is a vertex on the northeast corner of an obstacle which is bigger than 1×1. Suppose the obstacle is $k_1 \times k_2$, where k_1 is the length of the north (and south) side of the obstacle, and k_2 is the length of the east (and west) side of the obstacle. We know by the inductive hypothesis that the canonical shortest path from p_w goes either east or west along the north side of this obstacle, and since the vertex p has depth t we know that the canonical shortest path goes west. After having reached the corner, the canonical shortest path from p_w to s proceeds south. Thus, the vertex which is on the southwest corner of this obstacle has depth $l = t - 1 - (k_1 - 1) - k_2$. If we go from this vertex to p_s along the south side of the obstacle and then along the east side of the obstacle, then the depth of point p_s is at most $l + k_1 + (k_2 - 1) = t - 1$. Thus, in this case there is also a canonical path from p to s which goes south whenever possible.

∎

Lemma 7 *Consider adjacent vertices v and w in a city-block graph where v is north of w. In the northern region, without loss of generality, $d[v] = d[w] + 1$.*

Proof: The proof follows immediately from Lemma 6. ∎

Lemma 8 *Consider adjacent vertices v and w in a city-block graph where v is west of w. In the northern region, without loss of generality, $d[v] = d[w] \pm 1$.*

Proof: We prove the lemma by induction on the y-coordinate of the vertices in the northern region. If v and w have the same y-coordinate as s, then we know that $d[v] = d[w] + 1$ if s is east of v and $d[v] = d[w] - 1$ if s is west of w. Assume that the claim is true for vertices v and w with y-coordinate k. In the following we show that it is also true for vertices v and w with y-coordinate $k + 1$. We distinguish the case that there is no obstacle directly south of v and w from the case that there is an obstacle directly south of v or w.

Case 1: If there is no obstacle directly south of v and w, or there a 1×1 obstacle with u and w on the north side, the lemma follows by Lemma 7 and the induction assumption.

Case 2: If there is an obstacle directly south of v or w, then we assume without loss of generality that both v and w are on the boundary of the north side of the obstacle. (Note that v or w may, however, be at a corner of the obstacle.)

If the lemma does not hold it means that $d[v] = d[w]$ for two adjacent vertices v and w (because, in any graph, the d values for adjacent vertices can differ by at most one). This would also mean that all shortest paths from v to s must go through vertex v_w at the north-west corner of the obstacle and all shortest paths from w to s must go through vertex v_e at the north-east corner of the obstacle (v_w may be the same as v, and v_e may be the same as w). However, we next show that there is a grid point m on the boundary of the north side of the obstacle that has shortest paths through both v_e and v_w. The claim of Lemma 8 follows directly.

The distance x between m and v_w can be obtained by solving the following equation: $x + d[v_w] = (k - x) + d[v_e]$ where k is the length of the north side of the obstacle. The distance x is $(k + d[v_e] - d[v_w])/2$. Using the inductive hypothesis and Lemma 6, we know that if k is even then $|d[v_e] - d[v_w]|$ is even, and if k is odd then $|d[v_e] - d[v_w]|$ is odd. Thus the distance x is integral, and m exists in the graph.

■

3.6.2 The wavefront algorithm

The *wavefront algorithm* is based on BFS, but overcomes the inefficiency BFS has due to relocation cost. In this section, we first develop some preliminary concepts and results based on an analysis of breadth-first search in city-block graphs. We then present the wavefront algorithm, prove its correctness, and show that it runs in linear time.

Properties of BFS in city-block graphs

In city-block graphs, BFS can be viewed as exploring the graph in waves that expand outward from the start vertex s, much as waves expand from a pebble thrown into a pond. Figure 3.5 illustrates the wavefronts that can arise.

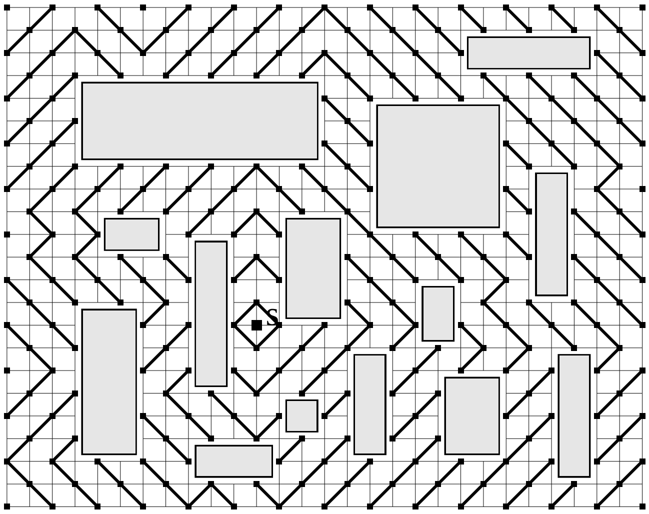


Figure 3.5: Environment explored by breath-first search, showing only "wavefronts" at odd distance to s.

A *wavefront* w can then be defined as an ordered list of explored vertices $\langle v_1, v_2, \ldots, v_m \rangle$, $m \geq 1$, such that $d[v_i] = d[v_1]$ for all i, and such that $\delta(v_i, v_{i+1}) \leq 2$ for all i. (As we shall prove in Lemma 9, the distance between adjacent points in a wavefront is always exactly equal to 2.) We call $d[w] = d[v_1]$ the distance of the wavefront.

There is a natural "successor" relationship between BFS wavefronts, as a wavefront at distance t generates a successor at distance $t + 1$. We informally consider a *wave* to be a sequence of successive wavefronts. Because of obstacles, however, a wave may split (if it hits an obstacle) or merge (with another wave, on the far side of an obstacle). Two wavefronts are *sibling* wavefronts if they each have exactly one endpoint on the same obstacle and if the waves to which they belong merge on the far side of that obstacle. The point on an obstacle where the waves first meet is called the *meeting point* m of the obstacle. In the northern region, meeting points are always on the north side of obstacles, and each obstacle has exactly one meeting point on its northern side. See Figure 3.6.

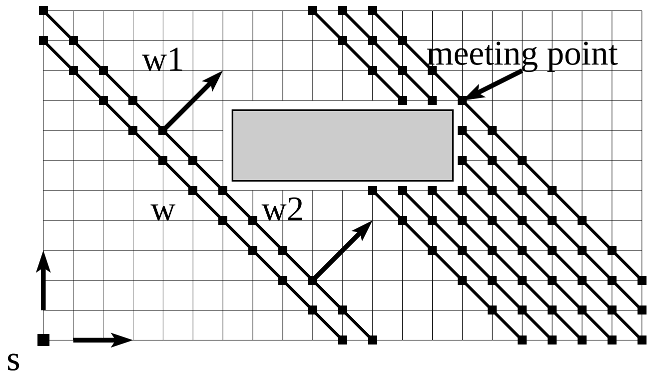


Figure 3.6: Splitting and merging of wavefronts along a corner of an obstacle. Illustration of meeting point and sibling wavefronts: w_1 and w_2 are sibling wavefronts which belong to different "waves." The waves merge at the meeting point.

Lemma 9 *A wavefront can only consist of diagonal segments.*

Proof: By definition a wavefront is a sequence of vertices at the same distance to s for which the distance between adjacent vertices is at most 2. It follows from Lemma 7 and 8 that neighboring points in the grid cannot be in the same wavefront. Therefore, the distance between adjacent vertices is exactly 2. Thus, the wavefront can only consist of diagonal segments. ■

We call the points that connect diagonal segments (of different orientation) of a wavefront *peaks* or *valleys*. In the northern region, a peak is a vertex on the wavefront that has a larger y-coordinate than the y-coordinates of its adjacent vertices in the wavefront, and a valley is a vertex on the wavefront that has a smaller y-coordinate than the y-coordinates of its adjacent vertices (see Figure 3.7).

The initial wavefront is just a list containing the start point s. Until a successor of the initial wavefront hits an obstacle, the successor wavefronts in the northern region consist of two diagonal segments connected by a peak. This peak is at the same x-coordinate for these successive wavefronts. Therefore, we say that the *shape* of the wavefronts does not change. In the northern region a wavefront can only have descendants that have a different shape if a descendant curls around the northern corners of an obstacle, or if it merges with another wavefront, or if it splits into other wavefronts. These descendants may then have more complicated shapes.

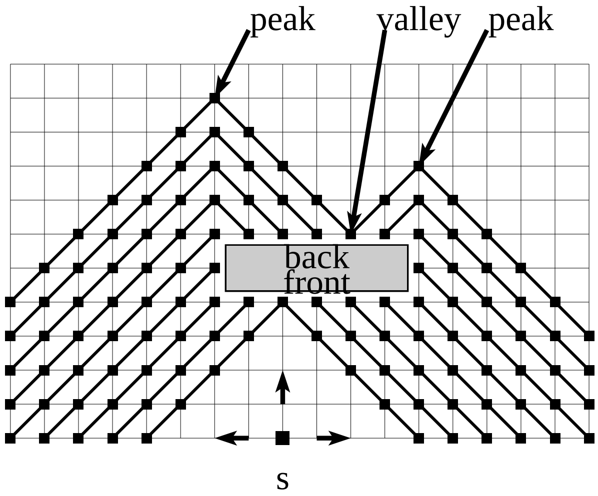


Figure 3.7: Shapes of wavefronts. Illustration of peaks and valleys, and front and back of an obstacle. The meeting point is the lowest point in the valley.

A wavefront w splits whenever its hits an obstacle. That is, if a vertex v_i in the wavefront is on the boundary of an obstacle, w splits into wavefronts $w_1 = \langle v_1, v_2, \ldots, v_i \rangle$ and $w_2 = \langle v_i, v_{i+1}, \ldots, v_m \rangle$. Wavefront w_1 propagates around the obstacle in one direction, and wavefront w_2 propagates around in the other direction. Eventually, some descendant wavefront of w_1 and some descendant wavefront of w_2 will have a common point on the boundary of the obstacle—the meeting point. The position of the meeting point is determined by the shape of the wave approaching the obstacle. (In the proof of Lemma 8, vertex m is a meeting point and we showed how to calculate its position once the length k of the north side of the obstacle and the shortest path distances of the vertices v_e and v_w at the north-east and north-west corners of the obstacle are known: the distance from v_w to the meeting point m is $(k + d[v_w] - d[v_e])/2$.)

In the northern region, the *front* of an obstacle is its south side, the *back* of an obstacle is its north side, and the *sides* of an obstacle are its east and west sides. A wave always hits the front of an obstacle first. Consider the shape of a wave before it hits an obstacle and its shape after it passes the obstacle. If a peak of the wavefront hits the obstacle (but not at a corner), this peak will not be part of the shape of the wave after it "passes" the obstacle. Instead, the merged wavefront may have one or two new peaks which have the same x-coordinates as the sides of the obstacle (see Figure 3.7). The merged wavefront has a valley at the meeting point on the boundary of the obstacle.

Description of the wavefront algorithm

The wavefront algorithm, presented in this section, mimics BFS in that it computes exactly the same set of wavefronts. However, in order to minimize relocation costs, the wavefronts may be computed in a different order. Rather than computing all the wavefronts at distance t before computing any wavefronts at distance $t+1$ (as BFS does), the wavefront algorithm will continue to follow a particular wave persistently, before it relocates and pushes another wave along.

We define *expanding* a wavefront $w = \langle v_1, v_2, \ldots, v_l \rangle$ as computing a set of zero or more successor wavefronts by looking at the set of all unexplored vertices at distance one from any vertex in w. Every vertex v in a successor wavefront has $d[v] = d[w] + 1$. The robot starts with vertex on one end of the wavefront and moves to all of its unexplored adjacent vertices. The robot then moves to the next vertex in the wavefront and explores its adjacent unexplored vertices. It proceeds this way down the vertices of the wavefront.

The following lemma shows that a wavefront of l vertices can be expanded in time $O(l)$.

Lemma 10 *A robot can expand a wavefront $w = \langle v_1, v_2, \ldots, v_l \rangle$ by traversing at most $2(l-1) + 2\lceil l/2 \rceil + 4$ edges.*

Proof: To expand a wavefront $w = \langle v_1, v_2, \ldots, v_l \rangle$ the robot needs to move along each vertex in the wavefront and find all of its unexplored neighbors. This can be done efficiently by moving along pairs of unexplored edges between vertices in w. These unexplored edges connect l of the vertices in the successor wavefront. This results in at most $2(l-1)$ edge traversals, since neighboring vertices are at most 2 apart. The successor wavefront might have $l+2$ vertices, and thus at the beginning and the end of the expansion (i.e., at vertices v_1 and v_l), the robot may have to traverse an edge twice. In addition, at any vertex which is a peak, the robot may have to traverse an edge twice. Note that a wavefront has at most $\lceil l/2 \rceil$ peaks. Thus, the total number of edge traversals is at most $2(l-1) + 2\lceil l/2 \rceil + 4$. ■

Since our algorithm computes exactly the same set of wavefronts as BFS, but persistently pushes one wave along, it is important to make sure the wavefronts are expanded correctly. There is really only one incorrect way to expand a wavefront and get something other than what BFS obtained as a successor: to expand a wavefront that is touching a meeting point before its sibling wavefront has merged with it. Operationally, this means that the wavefront algorithm is blocked in the following two situations:

(a) It cannot expand a wavefront from the side around to the back of an obstacle before the meeting point for that obstacle has been set (see Figure 3.8).

(b) It cannot expand a wavefront that touches a meeting point until its sibling has arrived there as well (see Figure 3.9).

A wavefront w_2 *blocks* a wavefront w_1 if w_2 must be expanded before w_1 can be safely expanded. We also say w_2 and w_1 *interfere*.

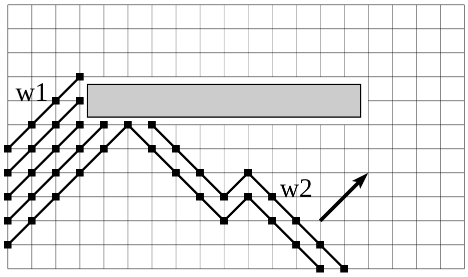


Figure 3.8: Blockage of w_1 by w_2. Wavefront w_1 has finished covering one side of the obstacle and the meeting point is not set yet.

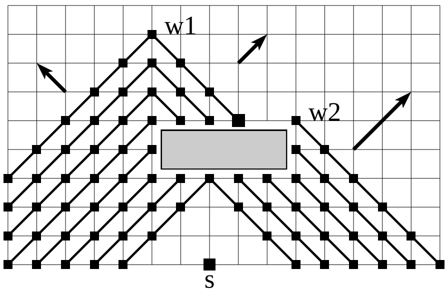


Figure 3.9: Blockage of w_1 by w_2. Wavefront w_1 has reached the meeting point on the obstacle, but the sibling wavefront w_2 has not.

A wavefront w is an *expiring* wavefront if its descendant wavefronts can never interfere with the expansion of any other wavefronts that now exist or any of their descendants. A wavefront w is an expiring wavefront if its endpoints are both on the front of the same obstacle; w will expand into the region surrounded by the wavefront and the obstacle, and then disappear or "expire." We say that a wavefront expires if it consists of just one vertex with no unexplored neighbors.

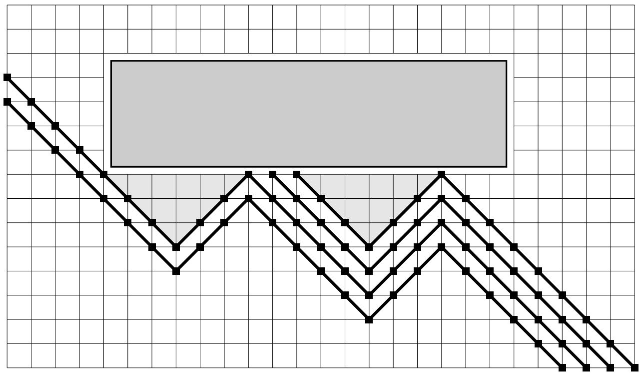


Figure 3.10: Triangular areas (shaded) delineated by two expiring wavefronts.

Procedure Wavefront-algorithm is an efficient optimally interruptible search algorithm that can be used to create an efficient piecemeal learning algorithm. It repeatedly expands one wavefront until it splits, merges, expires, or is blocked. The Wavefront-algorithm takes as an input a start point s and the boundary coordinates of the environment. It calls procedure Create-monotone-paths to explore four monotone paths (see Section 3.6.1) and define the four regions. Then procedure Explore-area is called for each region.

```
WAVEFRONT-ALGORITHM (s, boundary)
create monotone paths
For region = north, south, east, and west
    initialize current wavefront w := ⟨s⟩
    EXPLORE-AREA (w, region)
    take a shortest path to s
```

For each region we keep an ordered list L of all the wavefronts to be expanded. In the northern region, the wavefronts are ordered by the x-coordinate of their west-most point. *Neighboring* wavefronts are wavefronts that are adjacent in the ordered list L of wavefronts. Note that for each pair of neighboring wavefronts there is an obstacle on which both wavefronts have an endpoint.

Initially, we expand each wavefront in the northern region from its west-most endpoint to its east-most endpoint (i.e., we are expanding wavefronts in a "west-to-east" manner). The direction of expansion changes for the first time in the northern region when a wavefront is blocked by a wavefront to its west (the direction of expansion then becomes "east-to-west"). In fact, the direction of expansion changes each time a wavefront is blocked by a wavefront that is in the direction opposite of expansion. We introduce this notion of expanding wavefronts in either "west-to-east" or "east-to-west" directions in order to simplify the analysis of the algorithm.

We treat the boundaries as large obstacles. The north region has been fully explored when the list L of wavefronts is empty. Note that vertices on the monotone paths are considered initially to be unexplored, and that expanding a wavefront returns a successor that is entirely within the same region.

Each iteration of EXPLORE-AREA expands a wavefront. When EXPAND is called on a wavefront w, the robot starts expanding w from its current location, which is a vertex at one of the endpoints of wavefront w. It is often convenient, however, to think of EXPAND as finding the unexplored neighbors of the vertices in w in parallel.

Depending on what happens during the expansion, the successor wavefront can be split, merged, blocked, or may expire. Note that more than one of these cases may apply.

Procedures MERGE and SPLIT (see following pages) handle the (not necessarily disjoint) cases of merging and splitting wavefronts. Note that we use call-by-reference conventions for the wavefront w and the list L of wavefronts (that is, assignments to these variables within procedures MERGE and SPLIT affect their values in procedure EXPLORE-AREA). Each time procedure RELOCATE(w, dir) is called, the robot moves from its current location to the appropriate endpoint of w: in the northern region, if the direction is "west-to-east" the robot moves to the west-most vertex of w, and if the direction is "east-to-west," the robot moves to the east-most vertex of w.

Procedure RELOCATE(w, dir) can be implemented so that when it is called, the robot simply moves from its current location to the appropriate endpoint of w via a shortest path in the explored area of the graph. However, for analysis purposes, we assume that when RELOCATE(w, dir) is called the robot moves from its current location to the appropriate endpoint of w as follows.

- When procedure RELOCATE(w_s, dir) is called in line 5 of EXPLORE-AREA, the robot traverses edges between the vertices in wavefront w_s to get back to the appropriate endpoint of the newly expanded wavefront.

```
EXPLORE-AREA (w, region)
initialize list of wavefronts L := ⟨w⟩
initialize direction dir := west-to-east
Repeat
   EXPAND current wavefront w to successor wavefront w_s
   RELOCATE (w_s, dir)
   current wavefront w := w_s
   If w is a single vertex with no unexplored neighboring vertices
         Then
           remove w from ordered list L of wavefronts
           If L is not empty
                 Then
                   w := neighboring wavefront of w in direction dir
                   RELOCATE (w, dir)
         Else
           replace w by w_s in ordered list L of wavefronts
           If the second back corner of any obstacle(s)
                   has just been explored
                 Then set meeting points for those obstacle(s)
           If w can be merged with adjacent wavefront(s)
                 Then MERGE (w, L, region, dir)
           If w hits obstacle(s)
                 Then SPLIT (w, L, region, dir)
   If L not empty
         Then
           If  w is blocked by neighboring wavefront w′ in direction
                   D ∈ {west-to-east, east-to-west}
                 Then
                   dir := D
                   While w is blocked by neighboring wavefront w′
                         Do
                           w := w′
                           RELOCATE (w, dir)
Until L is empty
```

- When procedure Relocate(w_s, dir) is called in line 13 of Explore-area, the robot traverses edges along the boundary of an obstacle.

- When procedure Relocate(w_s, dir) is called in line 9 of Merge, the robot traverses edges between vertices in wavefront w to get to the appropriate endpoint of the newly merged wavefront.

- When procedure Relocate(w_s, dir) is called in line 30 of Explore-area, the robot traverses edges as follows. Suppose the robot is in the northern region and at the west-most vertex of wavefront w_0, and assume that w is to the east of w_0. Note that both w_0 and w are in the current ordered list of wavefronts L. Thus there is a path between the robot's current location and wavefront w which "follows the chain" of wavefronts between w_0 and w. That is, the robot moves from w_0 to w as follows. Let $w_1, w_2, \ldots, w_k$ be the wavefronts in the ordered list of wavefronts between w_0 and and w, and let $b_0, b_1, \ldots b_{k+1}$ be the obstacles separating wavefronts $w_0, w_1, \ldots, w_k, w$ (i.e., obstacle b_0 is between w_0 and w_1, obstacle b_1 is between w_1 and w_2, and so on). Then to relocate from w_0 to w, the robot traverses the edges between vertices of wavefront w_0 to get to the east-most vertex of w_0 which is on obstacle b_0. Then the robot traverses the edges of the obstacle b_0 to get to the west-point vertex of w_1, and then the robot traverses the edges between vertices in wavefront w_1 to get to the east-most vertex of w_1 which is on obstacle b_1. The robot continues traversing edges in this manner (alternating between traversing wavefronts and traversing obstacles) until it is at the appropriate end vertex of wavefront w.

```
Merge (w, L, region, dir)
1  remove w from list L of wavefronts
2  While there is a neighboring wavefront w' with which w can merge
3  Do
4      remove w' from list L of wavefronts
5      merge w and w' into wavefront w''
6      w := w''
7      put w in ordered list L of wavefronts
8  If w is not blocked
9      Then Relocate (w, dir)
```

Wavefronts are merged when exploration continues around an obstacle. A wavefront can be merged with two wavefronts, one on each end.

When procedure Split is called on wavefront w, we note that the wavefront is either the result of calling procedure Expand in line 4 of Explore-area or the result of calling procedure Merge in line 19 of Explore-area. Once wavefront w is split into $w_0, \ldots, w_n$, we update the ordered list L of wavefronts, and update the current wavefront.

```
SPLIT (w, L, region, dir)
split w into appropriate wavefronts w_0, ..., w_n in standard order
remove w from ordered list L of wavefronts
For i = 0 To n
    put w_i on ordered list L of wavefronts
    If dir = west-to-east
        Then  w:= w_0
        Else  w:= w_n
```

Correctness of the wavefront algorithm

The following theorems establish the correctness of our algorithm.

Theorem 8 *The algorithm* Explore-area *expands wavefronts so as to maintain optimal interruptibility.*

Proof: This is shown by induction on the distance of the wavefronts. The key observations are:

- There is a canonical shortest path from any vertex v to s which goes south whenever possible, but east or west around obstacles.
- A wavefront is never expanded beyond a meeting point.

We show that the algorithm maintains optimal interruptibility by knowing the canonical shortest path from any explored vertex to the start vertex s. We refer to this as the *shortest path property.* We show that the algorithm maintains the shortest path property by induction on the number of stages in the algorithm. Each stage of the algorithm is an expansion of a wavefront.

The shortest path property is trivially true when the number of stages $k = 1$. There is initially only one wavefront, the start point. Now we assume all wavefronts that exist just after the k-th stage satisfy the shortest path property, and we want to show that all wavefronts that exist just after the $k+1$-st stage also satisfy the shortest path property.

Consider a wavefront w in the k-th stage which the algorithm has expanded in the $k+1$-st stage to w_s. We claim that all vertices in w_s have shortest path length $d[w]+1$. Note that any vertex in w_s which is directly north of a vertex in w definitely has shortest path length $d[w]+1$. This is because there is a shortest path from any vertex v to s which goes south whenever possible, but if it is not possible to go south because of an obstacle, it goes east or west around the obstacle.

The only time any vertex v in w_s is not directly north of a vertex in w is when w is expanded around the back of an obstacle. This can only occur for a vertex that is either the west-most or east-most vertex of a wavefront in the north region. Without loss of generality we assume that v is the west-most point on w_s and v is on the boundary of some obstacle b. Note that w is expanded around the back of an obstacle only when the meeting point is determined. Because the algorithm only expands any wavefront until it reaches the meeting point of an obstacle, vertex v is not to the west of the meeting point. The algorithm knows that v has a shortest path from s that goes through v_c and along the obstacle to v. Thus the algorithm satisfies the shortest path property for the $k+1$-st stage. ∎

Theorem 9 *If the region is not completely explored, there is always a wavefront that is not blocked.*

Proof: We consider exploration in the north region. The key observations are:

- Neighboring wavefronts cannot simultaneously block each other.
- The east-most wavefront in the north region cannot be blocked by anything to its east, and the west-most wavefront in the north region cannot be blocked by anything to its west.

Thus the robot can always "follow a chain" of wavefronts to either its east or west to find an unblocked wavefront.

A neighboring wavefront is either a sibling wavefront or an expiring wavefront. An expiring wavefront can never block neighboring wavefronts. In order to show that neighboring wavefronts cannot simultaneously block each other, it thus suffices to show next that sibling wavefronts cannot block each other. We use this to show that we can always find a wavefront $\hat{w}$ which is not blocked. The unblocked wavefront $\hat{w}$ nearest in the ordered list of wavefronts L can be found by "following the chain" of blocked wavefronts from w to $\hat{w}$. By following the chain of wavefronts between w and $\hat{w}$ we mean that the robot must traverse the edges that connect the vertices in each wavefront between w and $\hat{w}$ in L and also the edges on the boundaries of the obstacles between these wavefronts. Note that neighboring wavefronts in list L each have at least one endpoint that lies on the boundary of the same obstacle.

Before we show that sibling wavefronts cannot block each other we need the following terminology. The first time an obstacle is discovered by some wavefront, we call the point that the wavefront hits the obstacle the *discovery* point. (Note that there may be more than one such point. We arbitrarily choose one of these points.) In the north region, we split up the wavefronts adjacent to each obstacle into an *east wave* and a *west wave*. We call the set of all these wavefronts which are between the discovery point and the meeting point of the obstacle in a west-to-east manner the west wave. We define the east wave of an obstacle analogously.

The discovery point of an obstacle b is always at the front of b. The wavefront that hits at b is split into two wavefronts, one of which is in the east wave and one of which is in the west wave of the obstacle. We claim that a descendent wavefront w_1 in the west wave and a descendant wavefront w_2 in the east wave cannot simultaneously block each other. Assume that the algorithm is trying to expand w_1 but that wavefront w_2 blocks w_1. Wavefront w_2 can only block w_1 if one of the following two cases applies. In both cases, we show that w_1 cannot also block w_2.

> **Case 1:** Wavefront w_1 is about to expand to the back of obstacle b, but both of the back corners of obstacle b have not been explored, and thus the meeting point has not been determined. Wavefront w_2 can only be blocked by w_1 if w_2 is either already at the meeting point of the obstacle or about to expand to the back of the obstacle. Since none of the back corners of obstacle b have been explored, neither of these two possibilities holds. Thus, wavefront w_1 does not block w_2.

Case 2: Wavefront w_1 has reached the meeting point at the back of b. Therefore, both back corners of the obstacle have been explored and w_1 is not blocking w_2.

We have just shown that if w_2 blocks w_1 then w_1 cannot also block w_2. Thus, the algorithm tries to pick w_2 as the nearest unblocked wavefront to w_1. However, w_2 may be blocked by its sibling wavefront w_3 on a different obstacle b'. For this case, we have to show that this sibling wavefront w_3 is not blocked, or that its sibling wavefront w_4 on yet another obstacle b'' is not blocked and so forth. Without loss of generality, we assume that the wavefronts are blocked by wavefronts towards the east. Proceeding towards the east along the chain of wavefronts will eventually lead to a wavefront which is not blocked—the east-most wavefront in the northern region. The east-most wavefront is adjacent to the initial monotone east-north path. Therefore, it cannot be blocked by a wavefront towards the east.

∎

Theorem 10 *The wavefront algorithm is an optimally interruptible piecemeal learning algorithm for city-block graphs.*

Proof: To show the correctness of a piecemeal algorithm that uses our wavefront algorithm for exploration with interruption, we show that the wavefront algorithm maintains the shortest path property and explores the entire environment.

Theorem 8 shows by induction on shortest path length that the wavefront algorithm mimics breadth-first search. Thus it is optimally interruptible.

Theorem 9 shows that the algorithm does not terminate until all vertices have been explored. Correctness follows. ∎

Efficiency of the wavefront algorithm

We now show the number of edges traversed by the piecemeal algorithm based on the wavefront algorithm is linear in the number of edges in the city-block graph.

We first analyze the number of edges traversed by the wavefront algorithm. Note that the robot traverses edges when procedures Create-monotone-paths, Expand, and Relocate are called. In addition, it traverses edges to get back to s between calls to Explore-area.

These are the only times the robot traverses edges. Thus, we count the number of edges traversed for each of these cases. In Lemmas 11 to 14, we analyze the number of edges traversed by the robot due to calls of RELOCATE. Theorem 11 uses these lemmas and calculates the total number of edges traversed by the wavefront algorithm.

Lemma 11 *An edge is traversed at most once due to relocations after a wavefront has expired (*RELOCATE *in line 13 of* EXPLORE-AREA*).*

Proof: Assume that the robot is in the northern region and expanding wavefronts in a west-to-east direction. Suppose wavefront w has just expired onto obstacle b (i.e., it is a single vertex with all of its adjacent edges explored). The robot now must relocate along obstacle b to its neighboring wavefront w' to the east. Note hat w' is also adjacent to obstacle b, and therefore the robot is only traversing edges on the obstacle b.

Note that at this point of exploration, there is no wavefront west of w which will expire onto obstacle b. This is because expiring wavefronts are never blocked, and thus the direction of expansion cannot be changed due to an expiring wavefront. So, when a wavefront is split and the direction of expansion is west-to-east, the robot always chooses the west-most wavefront to expand first. Thus, the wavefronts which expire onto obstacle b are explored in a west to east manner. Thus relocations after wavefronts have expired on obstacle b continuously move east along the boundary of this obstacle. ∎

Lemma 12 *An edge is traversed at most once due to relocations after wavefronts have merged (*RELOCATE *in line 9 of* MERGE*).*

Proof: Before a call to procedure MERGE, the robot is at the appropriate end vertex of wavefront w. Let's assume that the robot is in the northern region and expanding wavefronts in a west-to-east direction. Thus the robot is at the west-most vertex of wavefront w. Note that wavefront w can be merged with at most two wavefronts, one at each end, but only merges with the wavefront to the west of w actually cause the robot to relocate. Suppose wavefront w is merged with wavefront w' to its west to form wavefront w''. Then, if the resulting wavefront w'' is unblocked, procedure RELOCATE is called and the robot must traverse w'' to its west-most vertex (i.e., also the west-most vertex of w'). However, since wavefront w'' is unblocked, w'' can immediately be expanded and is not traversed again. ∎

Lemma 13 *At most one wavefront from the east wave of an obstacle is blocked by one or more wavefronts in the west wave. At most one wavefront from the west wave is blocked by one or more wavefronts in the east wave.*

Proof: Consider the west wave of an obstacle. By the definition of blocking, there are only two possible wavefronts in the west wave that can be blocked. One wavefront is adjacent to the back corner of the obstacle. Call this wavefront w_1. The other wavefront is adjacent to the meeting point of the obstacle. Call this wavefront w_2.

We first show that if w_1 is blocked then w_2 will not be blocked also. Then we also know that if w_2 is blocked then w_1 must not have been blocked. Thus at most one wavefront in the west wave is blocked.

If w_1 is blocked by one or more wavefronts in the east wave then these wavefronts can be expanded to the meeting point of the obstacle without interference from w_1. That is, wavefront w_1 cannot block any wavefront in the east wave, and thus there will be no traversals around the boundary of the obstacle until the east wave has reached the meeting point. At this point, the west wave can be expanded to the meeting point without any wavefronts in the east wave blocking any wavefronts in the west wave.

Similarly, we know that at most one wavefront from the west wave is blocked by one or more wavefronts in the east wave. ∎

Lemma 14 *An edge is traversed at most three times due to relocation after blockage (*RELOCATE *in line 30 of* EXPLORE-AREA*).*

Proof: Without loss of generality, we assume that the wavefronts are blocked by wavefronts towards the east. Proceeding towards the east along the chain of wavefronts will eventually lead to a wavefront which is not blocked, since the east-most wavefront is adjacent to the initial monotone east-north path.

First we show that any wavefront is traversed at most once due to blockage. Then we show that the boundary of any obstacle is traversed at most twice due to blockage. Note that pairs of edges connecting vertices in a wavefront may also be edges which are on the boundaries of obstacles. Thus any edge is traversed at most three times due to relocation after blockage.

We know from Theorem 9 that there is always a wavefront that is not blocked. Assume that the robot is at a wavefront w which is blocked by a wavefront to its east. Following the chain of wavefronts to the east leads to an unblocked wavefront w'. This results in one traversal of the wavefronts. Now this wavefront w' is expanded until it is blocked by some wavefront w''. Note that wavefront w'' cannot be to the west of w', since we know that the wavefront west of w' is blocked by w'. (We show in the proof of Theorem 9 that if w_1 blocks w_2 then w_2 does not block w_1.) The robot will not move to any wavefronts west of wavefront w' until a descendant of w' no longer blocks the wavefront immediately to its west. Once this is the case, then the west wavefront can immediately be expanded. Similarly, we go back through the chain of wavefronts, since - as the robot proceeds west - it expands each wavefront in the chain. Thus the robot never traverses any wavefront more than once due to blockage.

Now we consider the number of traversals, due to blockage, of edges on the boundary of obstacles. As wavefronts expand, their descendant wavefronts may still be adjacent to the same obstacles. Thus, we need to make sure that the edges on the boundaries of obstacles are not traversed too often due to relocation because of blockage. We show that any edge on the boundary of an obstacle is not traversed more than twice due to relocations because of blockage. That is, the robot does not move back and forth between wavefronts on different sides of an obstacle. Lemma 13 implies that each edge on the boundary of the obstacle is traversed at most twice due to blockage.

Thus, since the edges on the boundary of an obstacle may be part of the pairs of edges connecting vertices in a wavefront, the total number of times any edge can be traversed due to blockage is at most three. ∎

Theorem 11 *The wavefront algorithm is linear in the number of edges in the city-block graph.*

Proof: We show that the total number of edge traversals is no more than $15|E|$. Note that when the procedures Create-monotone-paths, Expand, and Relocate are called, the robot traverses edges in the environment. In addition, the robot traverses edges in the environment to get back to s after exploration of each of the four regions. These are the only times the robot actually traverses edges in the environment. Thus, to calculate the total number of edge traversals, we count the edge traversals for each of these cases.

The robot traverses the edges on the monotone paths *once* when it explores them, and *once* to get back to the start point. This is clearly at most $2|E|$ edge traversals. The robot walks back to s four times after exploring each of the four regions. Thus the number of edges traversed here is at most $4|E|$. The proof of Lemma 10 implies that the total number of edge traversals caused by procedure EXPAND is at most $2|E|$. We now only need to consider the edge traversals due to calls to procedure RELOCATE.

Procedure RELOCATE is called four times within EXPLORE-AREA and MERGE. The four calls are due to expansion (line 5 of EXPLORE-AREA), expiring (line 13 of EXPLORE-AREA), merging (line 9 of MERGE) and blocking (line 30 of EXPLORE-AREA). Relocations after expanding a wavefront results in a total of $|E|$ edge traversals. Lemma 11 shows that edges are traversed at most twice due to expiring wavefronts. Lemma 12 shows that edges are traversed at most once due to relocations after merges. Finally, Lemma 14 shows that edges are traversed at most three times due to relocations after blockage. Thus the total number of edge traversals due to calls of procedure RELOCATE is at most $7|E|$.

Thus the total number edges traversed by the wavefront algorithm is at most $15|E|$. A more careful analysis of the wavefront algorithm can improve the constant factor. ∎

Theorem 12 *A piecemeal algorithm based on the wavefront algorithm runs in time linear in the number of edges in the city-block graph.*

Proof: This follows immediately from Theorem 10 and Theorem 11. ∎

3.6.3 The ray algorithm

We now give another efficient optimally interruptible search algorithm, called the *ray algorithm*. The ray algorithm is a variant of DFS that always knows a shortest path back to s. This thus yields another efficient piecemeal algorithm for searching a city-block graph. This algorithm is simpler than the wavefront algorithm, but may be less suitable for generalization, because it appears more specifically oriented towards city-block graphs.

The ray algorithm also starts by finding the four monotone paths, and splitting the graph into four regions to be searched separately. The algorithm explores in a manner similar to depth-first search, with the following exceptions. Assume that it is operating in the northern region. The basic operation is to explore a northern-going "ray" as far as possible, and then to return to the start point of the ray. Along the way, side-excursions of one-step are made to ensure the traversal of east-west edges that touch the ray. Optimal interruptibility will always be maintained: the ray algorithm will not traverse a ray until it knows a shortest path to s from the base of the ray (and thus a shortest path to s from any point on the ray, by Lemma 6).

The high-level operation of the ray algorithm is as follows. (See Figure 3.11.) From each point on the (horizontal segments of the) monotone paths bordering the northern region, a north-going ray is explored. On each such ray, exploration proceeds north until blocked by an obstacle or the boundary of the city-block graph. Then the robot backtracks to the beginning of the ray and starts exploring a neighboring ray. As described so far, each obstacle creates a "shadow region" of unexplored vertices to its north. These shadow regions are explored as follows. Once the two back corners of an obstacle are explored, the shortest paths to the vertices at the back of an obstacle are then known; the "meeting point" is then determined. Once the meeting point for an obstacle is known, the shortest path from s to each vertex on the back border of the obstacle is known. The robot can then explore north-going rays starting at each vertex at the back border of the obstacle. There may be further obstacles that were all or partially in the shadow regions; their shadow regions are handled in the same manner.

We note that not all paths to s in the "search tree" defined by the ray algorithm are shortest paths; the tree path may go one way around an obstacle while the algorithm knows that the shortest path goes the other way around. However, the ray algorithm is nonetheless an optimally interruptible search algorithm.

Theorem 13 *The ray algorithm is a linear-time optimally interruptible search algorithm that can be transformed into a linear-time piecemeal learning of a city-block graph.*

Proof: This follows from the properties of city-block graphs proved in Section 3.6.1, and the above discussion. In the ray algorithm each edge is traversed at most a constant number of times. The linearity of the corresponding piecemeal learning algorithm then follows from Theorem 7. ∎

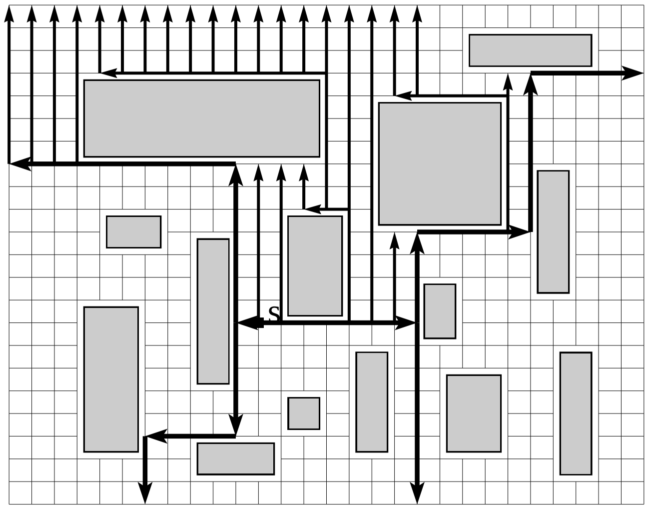


Figure 3.11: Operation of the ray algorithm.

3.7 Piecemeal learning of undirected graphs

For piecemeal learning of arbitrary undirected graphs, we again turn our attention to breadth-first search. As we mentioned earlier, standard BFS is efficient only when when the robot can efficiently switch or "teleport" from expanding one vertex to expanding another. In contrast, our model assumes a more natural scenario where the robot must physically move from one vertex to the next. We change the classical BFS model to a more difficult teleport-free exploration model, and give efficient *approximate* BFS algorithms where the robot does not move much further away from s than the distance from s to the unvisited vertex nearest to s. The teleport-free BFS algorithms we present never visit a vertex more than twice as far from s as the nearest unvisited vertex is from s.

Our techniques for piecemeal learning of arbitrary undirected graphs are inspired by the work of Awerbuch and Gallager [6, 7]. We observe that our learning model bears some similarity to the asynchronous distributed model. This similarity is surprising and has not been explored in the past.

Our main theorem for piecemeal learning of arbitrary undirected graphs is:

Theorem 14 *Piecemeal learning of an arbitrary undirected graph $G = (V, E)$ can be done in time $O(E + V^{1+o(1)})$.*

Proof: Following the Recursive-Strip algorithm, given in Section 3.7.3, the robot always knows a path from its current location back to the start vertex of length at most the radius of the graph. Thus RECURSIVE-STRIP is efficiently interruptible. The running time of this algorithm is $O(E + V 2^{O(\sqrt{\log V \log\log V})}) = O(E + V^{1+o(1)})$. By Theorem 7, this algorithm can be interrupted efficiently to give a piecemeal learning algorithm with running time $O(E + V^{1+o(1)})$.

∎

In the remainder of this section, we give three algorithms for piecemeal learning undirected graphs. We first give a simple algorithm that runs in $O(E + V^{1.5})$ time. We then give a modification of this algorithm that runs in $O((E + V^{1.5}) \log V)$ time. Although this algorithm has slightly slower running time, we are able to make it recursive, giving a third algorithm with almost linear running time: it achieves $O(E + V^{1+o(1)})$ running time. The most efficient previously known algorithm has $O(E + V^2)$ running time.

3.7.1 Algorithm STRIP-EXPLORE

This section describes an efficiently interruptible algorithm for undirected graphs with running time $O(E + V^{1.5})$. It is based on breadth-first search.

A *layer* in a BFS tree consists of vertices that have the same shortest path distance to the start vertex. A *frontier vertex* is a vertex that is incident to unexplored edges. A frontier vertex is *expanded* when the robot has traversed all the unexplored edges incident to it.

The traditional BFS algorithm expands frontier vertices layer by layer. In the teleport-free model, this algorithm runs in time $O(E + rV)$, since expanding all the vertices takes time $O(E)$, and visiting all the frontier vertices on layer i can be performed with a depth-first search of layers $1 \ldots i$ in time $O(V)$, and there are at most r layers. The procedure LOCAL-BFS describes a version of the traditional BFS procedure that has been modified for our teleport-free BFS model in two respects. First, the robot does not relocate to frontier vertices that have no unexplored edges. Second, it only explores vertices within a given distance-bound L of the given start vertex s. (The first modification, while seemingly straightforward, is essential for our analysis of STRIP-EXPLORE which uses LOCAL-BFS as a subroutine.) A procedure call of the form LOCAL-BFS(s, r), where s is the start vertex of the graph and r is its radius, would cause the robot to explore the entire graph.

Awerbuch and Gallager [6, 7] give a distributed BFS algorithm which partitions the network in *strips*, where each strip is a group of L consecutive layers. (Here L is a parameter to be chosen.) All vertices in strip $i-1$ are expanded before any vertices in strip i are expanded. Their algorithms use as a subroutine breadth-first type searches with distance L.

```
LOCAL-BFS(s, L)
For i = 0 To L − 1 Do
    let verts = all vertices at distance i from s
    For each u ∈ verts Do
            If u has any incident unexplored edges
            Then
               relocate to u
               traverse each unexplored edge
                       incident to u
relocate to s
```

Our algorithm, STRIP-EXPLORE, searches in strips in a new way. See Figure 3.12. The robot explores the graph in strips of width L. First the robot does LOCAL-BFS(s, L) to explore the first strip. It then explores the second strip as follows. Suppose there are k frontier vertices $v_1, v_2, \ldots, v_k$ in layer L; each such vertex is a *source vertex* for exploring the second strip. A naive way for exploring the second strip is for the robot for each i, to relocate to v_i, and then find all vertices that are within distance L of v_i by doing a BFS of distance-bound L from v_i within the second strip. The robot thus traverses a forest of k BFS trees of depth L, completely exploring the second strip. The robot then has a map of the BFS tree of depth L for the first strip and a map of the BFS forest for the second strip, enabling it to create a BFS tree of depth $2L$ for the first two strips. The robot continues, strip by strip, until the entire graph is explored.

The naive algorithm described above is inefficient, due to the overlap between the trees in the forest at a given level, causing portions of each strip to be repeatedly re-explored. The algorithm STRIP-EXPLORE presented below solves this problem by using the LOCAL-BFS procedure as the basic subroutine, instead of using a naive BFS. (See Figure 3.12.)

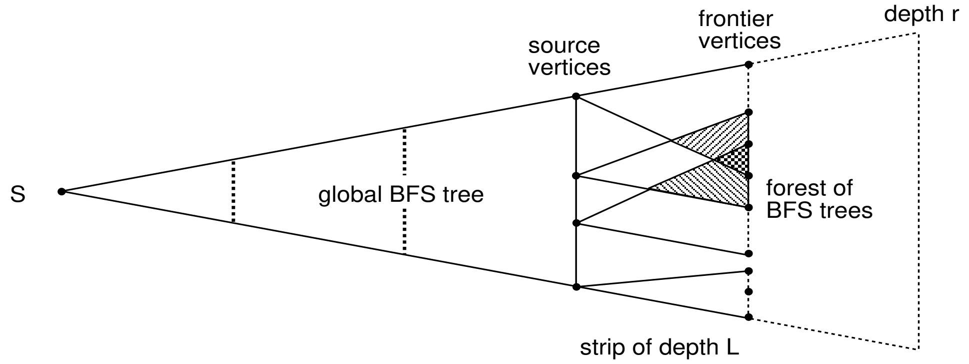


Figure 3.12: In the naive algorithm, the shaded areas are retraversed completely. In STRIP-EXPLORE, the shaded areas are passed through more than once only if necessary to get to frontier vertices.

```
STRIP-EXPLORE(s, L, r)
numstrips = ⌈r/L⌉
sources = {s}
For i = 1 To numstrips Do
        For each u ∈sources Do
            relocate to u
            LOCAL-BFS(u, L)
        sources = all frontier vertices
```

In STRIP-EXPLORE, the robot searches in a breadth-first manner, but ignores previously explored territory. The only time the robot traverses edges that have been previously explored is when moving to a frontier vertex it is about to expand. This results in retraversal of some edges in previously explored territory, but not as many as in the naive algorithm.

Theorem 15 STRIP-EXPLORE *runs in* $O(E + V^{1.5})$ *time.*

Proof: First we count edge traversals for relocating between source vertices for a given strip. For these relocations, the robot can mentally construct a tree in the known graph connecting these vertices, and then move between source vertices by doing a depth-first traversal of this tree. Thus the number of edge traversals due to relocations between source vertices for this strip is at most $2V$. Since there are $\lceil r/L \rceil$ strips, the total number of edge traversals due to relocations between source vertices is at most $\lceil \frac{r}{L} \rceil 2V \leq \left(\frac{r}{L} + 1\right) 2V = \frac{2rV}{L} + 2V$.

Now we count edge traversals for repeatedly executing the LOCAL-BFS algorithm. First,

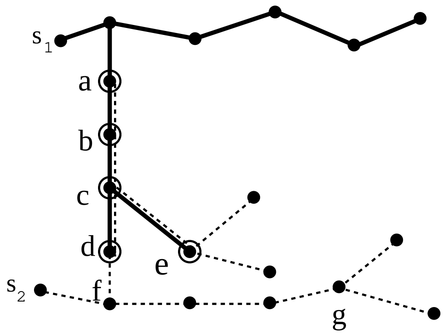


Figure 3.13: Contrasting BFS and Local-BFS: Consider a BFS of depth 5 from s_1, followed by a BFS of depth 5 from s_2. (The depth of the strip is $L = 5$.) The BFS from s_2 revisits vertices a, b, c, d, e. On the other hand, if the BFS from s_1 is followed by a LOCAL-BFS from s_2, then it only revisits d, c, e. After edge (f, d) is found, vertex e is a frontier vertex that needs to be expanded.

for the robot to expand all vertices and explore all edges, it traverses $2E$ edges. Next, each time the relocate in line 9 of procedure LOCAL-BFS is called, at most L edges are traversed. To account for relocations in line 6 of procedure LOCAL-BFS, we use the following scheme for "charging" edge traversals. Say the robot is within a call of the LOCAL-BFS algorithm. It has just expanded a vertex u and will now relocate to a vertex v to expand it. Vertex v is charged for the edges traversed to relocate from u to v. (We are only considering relocations within the same call of the LOCAL-BFS algorithm; relocations between calls of the LOCAL-BFS algorithm were considered above.) Source vertices are not charged anything. Moreover, the robot can always relocate from u to v by going from u to the source vertex of the current local BFS, and then to v, traversing at most $2L$ edges. Thus, each vertex is charged at most $2L$ when it is expanded. LOCAL-BFS never relocates to a vertex v unless it can expand vertex v (i.e., unless v is adjacent to unexplored edges). Thus, all relocations are charged to the expansion of some vertex, and the total number of edge traversals due to relocation is at most $2LV$.

Thus the total number of edge traversals is at most $2rV/L + 2V + 3LV + 2E$, which is $O(rV/L + LV + E)$. When L is chosen to be $\sqrt{r}$, this gives $O(E + V^{1.5})$ edge traversals. ∎

Procedure STRIP-EXPLORE, and the generalizations of it given in later sections, maintain that $\Delta \leq 2\delta$ at all times—the robot never visits a vertex more than twice as far from s as the nearest unvisited vertex is from s. The worst case is while exploring the second strip.

3.7.2 Iterative strip algorithm

We now describe Iterative-Strip, an algorithm similar to the Strip-Explore algorithm. It is an efficiently interruptible algorithm for undirected graphs inspired by Awerbuch and Gallager's [6] distributed iterative BFS algorithm. Although its running time of $O((V^{1.5} + E)\log V)$ is worse than the running time of Strip-Explore, its recursive version (described in Section 3.7.3) is more efficient. (It is not clear how to recursively implement Strip-Explore as efficiently, because the trees in a strip are not disjoint.)

With Iterative-Strip, the robot grows a global BFS tree with root s strip by strip, in a manner similar to Strip-Explore. Unlike Strip-Explore, here each strip is processed several times before it has correctly deepened the BFS tree by $\sqrt{r}$. We next explain the algorithm's behavior on a typical strip by describing how a strip is processed for the first time, and then for the remaining iterations.

```
Iterative-Strip(s, r)
For i = 1 To √r Do
   For each source vertex u in strip i Do
         relocate to u
         BFS from u to depth √r, but do not enter previously
            explored territory
   While there are any active connected components Iterate
         For each active connected component c Do
            Repeat
                  let v1, v2, v3, ... be active frontier vertices
                     exclusively in c with smallest depth among
                     active frontier vertices in c
                  relocate to each of v1, v2, v3, ..., and expand
            Until no more active frontier vertices exclusively in c
         determine new and active connected components
```

In the first iteration, a strip is explored much as in Strip-Explore. The robot explores a tree of depth $\sqrt{r}$ from each source vertex, by exploring in breadth-first manner from each source vertex, without re-exploring previous trees. Whenever the robot finds a *collision* edge connecting the current tree to another tree in the same strip, it does not enter the other tree. Unlike Strip-Explore, the robot does not traverse explored edges to get to the active frontier vertices on other trees. Therefore, after the first iteration, the trees explored are *approximate*

BFS trees that may have frontier vertices with depth less than $\sqrt{r}$ from some source vertex. These vertices become *active frontier vertices* for the next iteration. Thus, the current strip may not yet extend the global BFS tree by depth $\sqrt{r}$, so more iterations are needed until all frontier vertices are inactive and the global BFS tree is extended by depth $\sqrt{r}$ (see Figure 3.14).

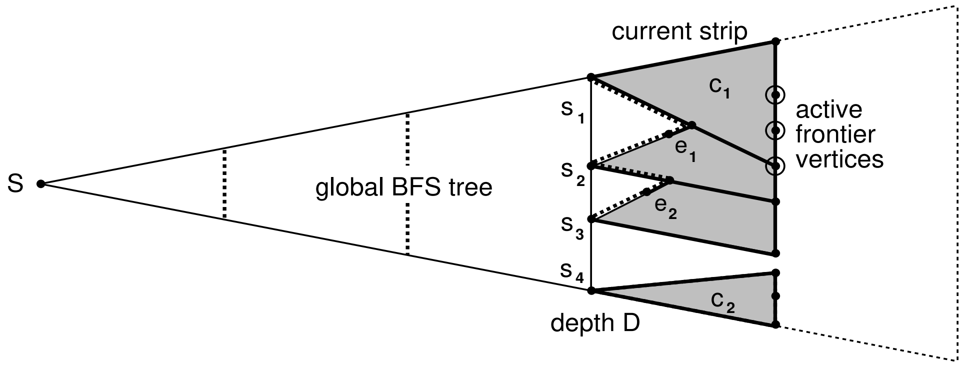


Figure 3.14: The iterative strip algorithm after the first iteration on the fourth strip. Two connected components c_1, c_2 have been explored. The collision edges e_1 and e_2 connect the first three approximate BFS trees. The dashed line shows how source vertices s_1, s_2, s_3 connect within the strip. There are three active frontier vertices with depth less than $D + \sqrt{r}$.

In the second iteration (see Figure 3.15), the robot uses the property that two trees connected by a collision edge form a connected component within the strip. (The graph to be explored is connected, and thus forms one connected component; but we refer to connected components of the explored portion of the graph contained within the strip.) The robot need not traverse any edges outside the current strip to relocate between these active frontier vertices in the same connected component. In the second and later iterations, the robot works on one connected component at a time.

The robot explores active frontier vertices in one connected component as follows. It computes (mentally) a spanning tree of the vertices in the current strip. This spanning tree lies within the strip. Let d be the least depth of any active frontier vertex in the component from a source vertex. It visits the vertices in the strip in an order determined by a DFS of the spanning tree. As it visits active frontier vertices of depth d, it expands them. It then recomputes the spanning tree (since the component may now have new vertices) and again traverses the tree, expanding vertices of the appropriate next depth d'. Traversing a collision edge does not add the new vertex to the tree, since this vertex has been explored before. This process continues

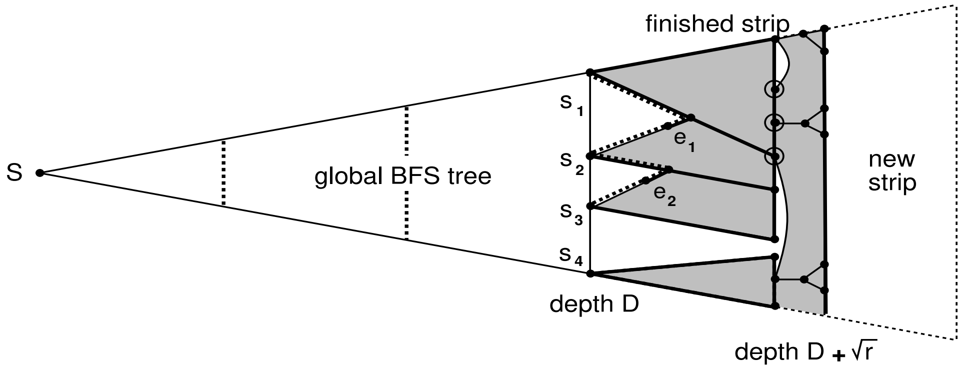


Figure 3.15: The iterative strip algorithm after the second iteration. Now the circled vertices which were active frontier vertices at the beginning of the iteration are expanded. One of the expansions resulted in a collision edge. Now the strip consists of only one connected component (shaded area). There are six frontier vertices which become source vertices of the next strip. All frontier vertices have depth $D + \sqrt{r}$.

(at most $\sqrt{r}$ times) until no active frontier vertex in the connected component has distance less than $\sqrt{r}$ from some source vertex in the component.

The robot handles each connected component in turn, as described above. In the next iteration it combines the components now connected by collision edges, and explores the new active frontier vertices in these combined components. Lemma 15 states that at most $\log V$ iterations cause all frontier vertices to become not active. That is, all frontier vertices are depth $\sqrt{r}$ from the source vertices of this strip. These frontier vertices are the new sources for the next strip.

Lemma 15 *At most* $\log V$ *iterations per strip are needed to explore a strip and extend the global BFS tree by depth* $\sqrt{r}$.

Proof: If there are initially l source vertices, then after the first iteration there are at most l connected components. If a component does not collide with another active component, then it will have no active frontier vertices for the next iteration. The only active components in the next iteration are those that have collided with other components, and thus, each iteration halves the number of components with active frontier vertices. After at most $\log V$ iterations there is no connected component with active frontier vertices left. The robot then has a complete map of the current strip and of the global BFS tree built in previous strips, so it can combine this information and extend the global BFS tree by depth $\sqrt{r}$. ∎

Theorem 16 ITERATIVE-STRIP *runs in time* $O((E + V^{1.5}) \log V)$.

Proof: We first count the number of edge traversals within a strip. Let V_i and E_i be the number of vertices and edges explored in strip i. For each component, vertices of distance t from some source vertex are expanded by computing a spanning tree of the component, doing a DFS of the spanning tree, and expanding all vertices of distance t from some source vertex (line 9). At each iteration (line 5), components are disjoint, so relocating to all vertices in the strip of distance exactly t takes at most $O(V_i)$ edge traversals. Thus, in one iteration, relocating to all vertices in the strip within distance $\sqrt{r}$ takes at most $O(\sqrt{r}V_i)$ edge traversals. Moreover, note that in order for the robot to expand each vertex, it traverses at most $O(E_i)$ edges. Thus, the total number of edge traversals for strip i in one iteration is $O(E_i + \sqrt{r}V_i)$. Combining this with Lemma 15, the total number of edge traversals within strip i to completely explore strip i takes $O((E_i + \sqrt{r}V_i) \log V)$ edge traversals.

Now we count edge traversals for relocating between source vertices in strip i. As in the proof of Theorem 15, in each iteration the robot traverses at most $2V$ edges to relocate between source vertices. Since there are at most $\log V$ iterations, this results in $2V \log V$ edge traversals between source vertices to explore strip i. Thus, the total number of edge traversals to explore strip i is $O((E_i + \sqrt{r}V_i) \log V + 2V \log V)$. Summing over the $\sqrt{r}$ disjoint strips gives $O((E + \sqrt{r}V) \log V + 2V\sqrt{r} \log V) = O((E + \sqrt{r}V) \log V) = O((E + V^{1.5}) \log V)$. ∎

3.7.3 A nearly linear time algorithm for undirected graphs

This section describes an efficiently interruptible algorithm RECURSIVE-STRIP, which gives a piecemeal learning algorithm with running time $O(E + V^{1+o(1)})$. RECURSIVE-STRIP is the recursive version of ITERATIVE-STRIP; it provides a recursive structure that coordinates the exploration of strips, of approximate BFS trees, and of connected components in a different manner. The robot still, however, builds a global BFS tree from start vertex s strip by strip. The robot expands vertices at the bottom level of recursion.

In RECURSIVE-STRIP, the depth of each strip depends on the level of recursion (see Figure 3.16). If there are k levels of recursion, then the algorithm starts at the top level by splitting the exploration of G into r/d_{k-1} strips of depth d_{k-1}. Each of these strips is split into d_{k-1}/d_{k-2}

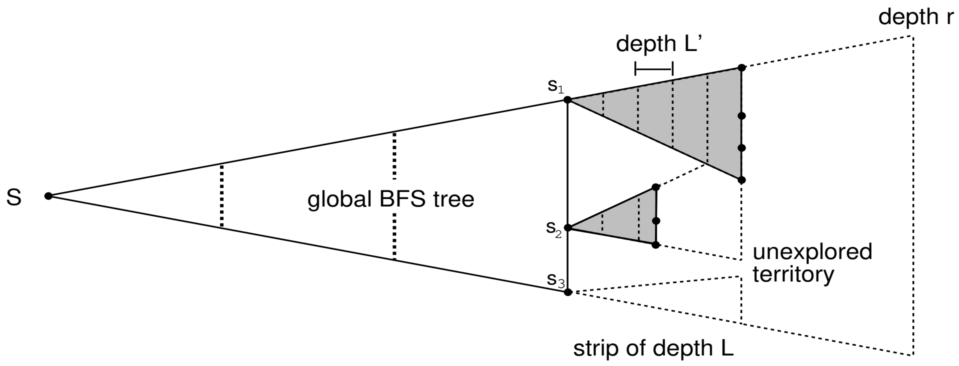


Figure 3.16: The recursive strip algorithm processing an approximate BFS tree from source vertex s_2 to depth $d_{k-1} = L$. Recursive calls within the tree are of depth $d_{k-2} = L'$.

searches of strips of depth d_{k-2}, etc. We have $r = d_k > d_{k-1} > \ldots > d_1 > d_0 = 1$.

Each recursive call of the algorithm is passed a set of source vertices *sources*, the *depth* to which it must explore, and a set T of all vertices in the strip already known to be less than distance *depth* from one of the sources. The robot traverses all edges and visits all vertices within distance *depth* of the sources that have not yet been processed by other recursive calls at this level. Recursive-Strip($\{s\}, r, \{s\}$) is called to explore the entire graph.

At recursion level i, the algorithm divides the exploration into strips and processes each strip in turn, as follows. Suppose the strip has l source vertices $v_1, \ldots, v_l$. The strip is processed in at most $\log l = O(\log V)$ iterations. In each iteration, the algorithm partitions T into maximal sets $T_1, T_2, \ldots, T_k$ such that each set is known to be connected within the strip. Let S_c denote the set of source vertices in T_c. A DFS of the spanning tree of the vertices T gives an order for the source vertices in $S_1, S_2, \ldots, S_k$; this spanning tree is used for efficient relocations between these source vertices. Note that all source vertices are known to be connected through the spanning tree of the vertices in T, but they might not be connected within the substrips. Since relocations between the vertices in S_c in the next level of recursion use a spanning tree of T_c, for efficiency the vertices of T_c must be connected within the strip. After partitioning the vertices into connected components within the strip, for each connected component T_c, the robot relocates (along a spanning tree) to some arbitrary source vertex in S_c. It then calls the algorithm recursively with S_c, the depth of the strip, and the vertices T_c which are connected to the sources S_c within the strip.

```
RECURSIVE-STRIP(sources, depth, T)
If depth = 1
   Then
         let v_1, v_2, ..., v_k be the depth-first ordering of sources
            in spanning tree
         For i = 1 To k Do
            relocate to v_i
            If v_i has adjacent unexplored edges
                  Then traverse v_i's incident edges
         T = T ∪ {newly discovered vertices}
         Return
Else
   determine next depth
   number-of-strips ← depth/next-depth
   For i = 1 To number-of-strips Do
         determine set of source vertices
         For j = 1 To number-of-iterations Do
            partition vertices in T into maximal sets T_1, T_2, ..., T_k
                  such that vertices in each T_c are known to be
                  connected within strip i
            For each T_c in suitable order Do
                  let S_c be the source vertices in T_c
                  relocate to some source s ∈ S_c
                  RECURSIVE-STRIP(S_c, next-depth, T_c)
                  T = T ∪ T_c
relocate to some s ∈ sources
Return
```

The remaining iterations in the strip combine the connected components until the strip is finished. Then the robot continues with the next strip in the same level of recursion. Or, if it finished the last strip, it relocates to its starting position and returns to the next higher level of recursion.

Theorem 17 RECURSIVE-STRIP *runs in time* $O(E + V^{1+o(1)})$.

Proof: At a particular call of RECURSIVE-STRIP, there are 4 places the robot traverses edges:

1. expansion of vertices in line 7
2. relocating to sources in lines 5 and 19
3. relocations due to recursive calls in line 20
4. relocation back to a beginning source vertex in line 22

We count edge traversals for each of these cases. First we give some notation. We consider the top level of recursion to be a level-k recursive call, and the bottom level of recursion to be a level-0 recursive call. For a particular level-i call of RECURSIVE-STRIP, let C_i denote the number of edge traversals due to relocations, and let E_i denote the number of distinct edges that are traversed due to relocation. Let V_i denote the number of vertices incident to these edges and whose incident edges are all known at the end of this call. Let ρ_i be a uniform upper bound on C_i/V_i. Thus, if the depth of recursion is k then the total number of edge traversals is bounded by $O(V\rho_k)$.

First we observe that each vertex is expanded at most once, so there are at most $O(E + V)$ edge traversals due to exploration at line 7 in the bottom level of recursion.

For a level-i call, we count the number of edge traversals for relocation between source vertices (lines 5 and 19). Since all the source vertices in the call are connected by a tree of size $O(V_i)$, relocating to all source vertices at the start of one strip takes $O(V_i)$ edge traversals. With d_i/d_{i-1} strips and $\log V$ iterations per strip, there are $V_i \log V \frac{d_i}{d_{i-1}}$ edge traversals for relocations between source vertices.

We now count traversals for recursive calls (line 20) within a level-i call. Note that our algorithm avoids re-exploring previously explored edges. Thus, for a level-i call, when working on a particular strip l, for each iteration within this strip, the sets of vertices whose edges are explored in each recursive call are disjoint. Suppose that, in this strip, in one iteration the procedure makes k recursive calls, each at level $i-1$. Then let $C_{i-1}^{(j)}$, $1 \leq j \leq k$, denote the number of edge traversals due to relocations resulting from the j-th recursive call, and let $V_{i-1}^{(j)}$ denote the number of vertices adjacent to these edges. Furthermore, let $V_{l,i}$ denote the number of vertices which are in strip l of this procedure call at recursion level i. Then we would like first to calculate $\sum_{j=1}^{k} C_{i-1}^{(j)}$, which is the number of edge traversals due to relocation in recursive calls in one iteration within this strip. This is at most $\sum_{j=1}^{k} \rho_{i-1} V_{i-1}^{(j)} = \rho_{i-1} \sum_{j=1}^{k} V_{i-1}^{(j)}$. Since the recursive calls are disjoint, $\sum_{j=1}^{k} V_{i-1}^{(j)} = V_{l,i}$, and thus the number of edge traversals due to relocations in recursive calls in one iteration within this strip is at most $\rho_{i-1} V_{l,i}$. Finally, since there are $\log V$ iterations in each strip, and all strips are disjoint from each other, the number of edge traversals due to recursive calls is at most $\rho_{i-1} V_i \log V$.

Finally, note that we relocate once at the end of each procedure call of RECURSIVE-STRIP (see line 22). This results in at most V_i edge traversals.

Thus, the number of edge traversals due to relocation (not including relocations for expanding vertices) is described by the recurrence $C_i \leq V_i \log V \frac{d_i}{d_{i-1}} + \rho_{i-1} V_i \log V + V_i$. Normalizing by V_i, we get the following recurrence:

$$\rho_i = \left(\frac{d_i}{d_{i-1}} + \rho_{i-1}\right) \log V + O(1)$$

Solving the recurrence for ρ_k gives:

$$\begin{aligned} \rho_k &\leq \left(\frac{d_k}{d_{k-1}}\right) \log V + \left(\frac{d_{k-1}}{d_{k-2}}\right) \log^2 V + \ldots + \left(\frac{d_1}{d_0}\right) \log^k V + \rho_0 \log^k V + \sum_{i=0}^{k-1} \log^i V \\ &\leq \left(\frac{d_k}{d_{k-1}}\right) \log V + \left(\frac{d_{k-1}}{d_{k-2}}\right) \log^2 V + \ldots + \left(\frac{d_1}{d_0}\right) \log^k V + O(\log^k V) \end{aligned}$$

We note that $\rho_0 = O(1)$, since at the bottom level, if there are V' vertices expanded, then the number of edge traversals due to relocation is $O(V')$. The product of the first k terms in the recurrence is $\frac{d_k}{d_0}(\log V)^{(k+1)k/2} = r(\log V)^{(k+1)k/2}$. We choose $d_{k-1}, d_{k-2}, \ldots$ by setting each of the first k terms equal to the k-th root of this product. (Note that this also specifies how to calculate depth d_{i-1} from depth d_i.) Substituting, we get:

$$\rho_k \leq k r^{1/k} (\log V)^{(k+1)/2} + O(\log^k V).$$

We find the value of k that minimizes this by taking the logarithm and differentiating with respect to k. Choosing $k = \left(\frac{\log V}{\log \log V}\right)^{1/2}$ and simplifying gives us $\rho_k \leq 2^{O(\sqrt{\log V \log \log V})}$, and thus C_k is at most $V 2^{O(\sqrt{\log V \log \log V})}$, which is $V^{1+o(1)}$. Adding the edge traversals for relocation to the edge traversals for expansion of vertices gives us $O(E + V^{1+o(1)})$ edge traversals total. ∎

3.8 An Application to Treasure Hunting

We now consider an application of our algorithms to the problem of finding a treasure (or a lost child, or a particular landmark) in an unknown, potentially infinite graph $G = (V, E)$. If the robot searching for the treasure knows that the treasure is close to its start location, it should explore in a manner such that it does not get too far away from this location.

We give the procedure Treasure-Search, which uses the Recursive-Strip algorithm as a subroutine. If the treasure is distance δ_T away from the source vertex, this algorithm maintains the condition that the robot is never further from the source than Δ, where $\Delta \leq \delta_T + o(\delta_T)$. Following procedure Treasure-Search, the robot traverses $O(E + V^{1+o(1)})$ edges, where E and V are the total number of distinct edges and vertices within radius Δ from the source.

The robot explores the graph for the treasure in phases. In each phase, the size of the strip to be explored changes. The change at phase i depends on $\epsilon_i = 1/\sqrt{i}$. Initially, the robot explores the graph out to distance $r_1 = 1 + \epsilon_1$. Next, the robot extends its exploration by a factor of $1+\epsilon_2$. That is, the size of the next strip is $(1+\epsilon_1)(1+\epsilon_2)-(1+\epsilon_1)$, and at the end of the second phase, the robot has learned the graph out to distance $r_2 = (1+\epsilon_1)(1+\epsilon_2)$. After extending the next strip, the robot has learned the graph out to distance $r_3 = (1+\epsilon_1)(1+\epsilon_2)(1+\epsilon_3)$, and so on. In each phase i, the robot initially calls Recursive-Strip from each of the source vertices (vertices at distance r_{i-1}). When the robot finds collision edges, it does not re-explore edges. Thus, within each phase, it may take up to $\log V$ iterations (as in Iterative-Strip and Recursive-Strip) before it has explored the entire strip.

```
TREASURE-SEARCH(s)
i = 0
r_0 = 1
Do until treasure is found
    i = i + 1
    ε_i = 1/√i
    r_i = r_{i-1} · (1 + ε_i)
    If i = 1
        Then
            RECURSIVE-STRIP({s}, r_1, {s})
        Else
            let T be be the set of source vertices distance
                r_{i-1} away from s
            For j = 1 To number-of-iterations Do
                partition vertices in T into maximal sets T_1, ..., T_k
                    such that vertices in each T_c are known to be
                    connected within strip i
                For each T_c in suitable order Do
                    let S_c be the source vertices in T_c
                    relocate to some source s ∈ S_c
                    RECURSIVE-STRIP(S_c, next-depth, T_c)
                    T = T ∪ T_c
```

Lemmas 16 and 17 bound the number of phases in the TREASURE-SEARCH procedure. Using Lemma 16, we can show that the robot does not get too far away from the source vertex, and using Lemma 17, we can bound the number of edges the robot traverses.

Lemma 16 *The number of phases in* TREASURE-SEARCH *is at least* $\log \delta_T$*.*

Proof: Since $\epsilon_1 > \epsilon_2 > \epsilon_3 \ldots$, we know that, for any j, $(1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_j) \leq (1+\epsilon_1)^j$. Thus, if we let j be the smallest number such that $(1+\epsilon_1)^j \geq \delta_T$, then we know that the number of phases i to reach the treasure at δ_T is at least j. Since $\epsilon_1 = 1$, we have $2^j \geq \delta_T$, or $j \geq \log \delta_T$. ∎

Lemma 17 *The number of phases in* TREASURE-SEARCH *is at most* $4 \ln^2 \delta_T + 1$*.*

Proof: A treasure at depth $\delta_T = 1$ is found in the first phase, so we consider only $\delta_T > 1$. We know that for any j, $(1+\epsilon_j)^j \leq (1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_j)$. Thus, if $(1+\epsilon_j)^j \geq \delta_T$, we know that the number of phases i is at most j. So we show the lemma by showing that $(1+\epsilon_{4\ln^2\delta_T})^{4\ln^2\delta_T} \geq \delta_T$. Equivalently, we would like to show $4\ln^2\delta_T \ln(1+\epsilon_{4\ln^2\delta_T}) = 4\ln^2\delta_T \ln(1+\frac{1}{2\ln\delta_T}) \geq \ln\delta_T$.

For $|x| < 1$, using a Taylor expansion, we have $\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots$. For $0 < x < 1$, we have $\ln(1+x) > x - \frac{x^2}{2}$. So $4\ln^2\delta_T \ln(1+\frac{1}{2\ln\delta_T}) > (4\ln^2\delta_T)(\frac{1}{2\ln\delta_T} - \frac{1}{8\ln^2\delta_T}) = 2\ln\delta_T - 1/2$, which is at least $\ln\delta_T$ for $\delta_T \geq 2$. ∎

Theorem 18 *The robot is never further than $\delta_T + \delta_T/\sqrt{\log\delta_T}$ from the source vertex.*

Proof: Let Δ be the furthest distance the robot gets from the source vertex. Let i be the number of phases that need to be explored to get out to depth δ_T. Then, $\Delta - \delta_T$ is at most the depth of the strip in the i-th phase. That is, $\Delta - \delta_T \leq (1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_i) - (1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_{i-1}) = (1+\epsilon_1)(1+\epsilon_2)\ldots(1+\epsilon_{i-1})\epsilon_i < \delta_T\epsilon_i$. Lemma 16 shows that the total number of strips explored is at least $\log\delta_T$. Thus, ϵ_i is at most $1/\sqrt{\log\delta_T}$, and $\Delta \leq \delta_T + \delta_T/\sqrt{\log\delta_T} = \delta_T + o(\delta_T)$. ∎

Theorem 19 *Procedure* TREASURE-SEARCH *traverses at most $O(E + V^{1+o(1)})$ edges, where E and V are the total number of distinct edges and vertices within radius Δ from the source.*

Proof: Since the edges in the different phases are disjoint, the number of edges traversed, ignoring relocations between source vertices in line 16, is at most $O(E + V^{1+o(1)})$. To get between source vertices in line 16, a spanning tree of the known vertices can be used. (Note that for recursive calls of RECURSIVE-STRIP, the algorithm relocates between source vertices using the vertices connected within the appropriate strip.) By Lemma 17, we know the number of phases is at most $4\ln^2\delta_T + 1$, and in each phase it may take up to $\log V$ iterations to explore the entire strip. Thus there are an additional $(4\ln^2\delta_T + 1)V\log V$ edge traversals due to relocations between source vertices, and this gives a total of $O(E + V^{1+o(1)})$ edge traversals for the entire TREASURE-SEARCH procedure. ∎

3.9 Conclusions

We have presented an efficient $O(E + V^{1+o(1)})$ algorithm for piecemeal learning of arbitrary, undirected graphs. For the special case of city-block graphs, we have given two linear time algorithms. We leave as open problems finding linear time algorithms (if they exist) for the piecemeal learning of:

- grid graphs with non-convex obstacles,
- other tesselations, such as triangular tesselations with triangular obstacles, and
- more general classes of graphs, such as the class of planar graphs.
- arbitrary, undirected graphs

CHAPTER 4

Learning-based algorithms for protein motif recognition

4.1 Introduction

One of the most important problems in computational biology is that of predicting how a protein will fold in three dimensions when we only have access to its one-dimensional amino acid sequence. Structure prediction has practical importance, as the biological function of a protein depends upon its structure or fold. Unfortunately, determining the three-dimensional structure of a protein is very difficult. Experimental approaches such as NMR and X-ray crystallography are expensive and time-consuming (they can take years), and often do not work at all. Therefore, computational techniques that predict protein structure based on already available sequence data can help speed up the understanding of protein functions.

An important first step in tackling the protein folding problem is a solution to the *structural motif recognition problem*: given a known local three-dimensional structure, or *motif*, determine whether this motif occurs in a given amino acid sequence, and if so, in what positions. In this chapter, we focus on a special type of α-helical motif, known as the coiled coil motif (see section 4.2), although the techniques presented can be applied to other motifs as well.

Most approaches to the motif recognition problem work only for motifs that are already well-studied — that is, they are known to occur in many sufficiently diverse proteins. This knowledge usually comes from biologists who have studied many examples of the motif. However, there are many motifs for which only a small subset of examples are known, and this subset is often not rich enough to be representative of the motif. Thus, for lack of data, current prediction methods ranging from straightforward sequence alignments to more complicated methods such as those based on profiles of the motifs often fail to successfully identify such motifs.

For example, in the case of the coiled coil motif, most known instances are 2-stranded coiled coils (i.e, coiled coils consisting of 2 α-helices). As a result, known prediction algorithms work well for predicting 2-stranded coiled coils [14, 13, 12, 42, 58, 63], but do not work as well for the related 3-stranded coiled coil motif (i.e., coiled coils consisting of 3 α-helices), due to the lack of known 3-stranded coiled coil sequences. That is, for 3-stranded coiled coils, these algorithms have a large amount of overlap between the scores for sequences that do not contain coiled coils and sequences that do.

Our results

In this chapter, we use learning theory to improve existing methods for protein structural motif recognition, particularly in the case where only a few examples of the motif are known. Our main result is a linear-time learning algorithm that uses information obtained from a database of sequences of one motif to make predictions about a related or similar motif.

The problem we explore can be viewed as a concept learning problem, where the algorithm is given labeled and unlabeled examples, and its goal is to find a concept which gives labels to all the examples. Unlike many concept learning frameworks, this problem is not completely supervised—this type of learning, which we refer to as *semi-supervised learning*, is often necessary in real-life learning problems. We find this to be true in our test domain, where our goal is to identify sequences that contain coiled coils from a set of protein sequences which may or may not contain coiled coils. In particular, we are interested in recognizing both 2- and 3-stranded coiled coils. Unfortunately, the majority of data we have is comprised of 2-stranded coiled coils. In addition, although many biologists are interested in 3-stranded coiled coils, there is little well analyzed data available on them. Thus, because of the lack of data and current biological knowledge, supervised learning (i.e., the algorithm is given a large enough set of examples of both 2- and 3-stranded coiled coils on which to train) is not currently feasible for our problem, and semi-supervised or even unsupervised learning (with no labeled examples) is the only type of learning which is possible. At first glance, this learning problem seems like a challenging problem, since we are trying to come up with an algorithm which generalizes the data we have for 2-stranded coiled coils to also pick out 3-stranded coiled coils. However, we show empirically that for our test domain, semi-supervised learning gives excellent results. In particular, we have tested our program and show that our algorithm's performance is substantially better than that of previously known algorithms for recognizing 3-stranded coiled coils.

Our algorithm starts with an original database of a *base motif,* and the goal is to develop a more general database of a *target motif,* which is related to the base motif in structure. (The target motif includes the base motif as a special case.) In other words, we would like to convert a good predictor for the base motif into a good predictor for the target motif. Our algorithm has the following key features:

- The algorithm iteratively scans a large database of test sequences to find sequences that are presumed to fold into the target motif. The selected sequences are then used to update the parameters of the algorithm; these updates affect the performance of the algorithm in the next iteration.
- In each iteration, the algorithm scores all the sequences based on its current estimates of the parameters and the theoretical framework developed in [12].
- In each iteration, the algorithm uses randomness to select which sequences are presumed to fold into the target motif.
- The selected sequences are used in the beginning of the next iteration to update the parameters of the algorithm in a Bayesian-like weighting scheme.

There are several ways in which our iterative algorithm is kept running in a "safe" fashion, without increasing the false positive rate by incorporating sequences into the final database that do not fold into the motif. First, we begin with a mathematically sound scoring subroutine, that experimentally has a low false positive rate. Second, our method of computing likelihoods ensures that only a certain fraction of all residues are scored as positive examples of the motif (see section 4.3). Finally, while evaluating our program, we run the program with sequences that are known not to contain coiled coils, and this has helped us determine when the algorithm is performing well.

This methodology does not appear to have been explored much in the biological literature. Although a few papers have dealt with iterative algorithms [73, 3, 46, 36], they do not use randomness and weighting for updating of parameters. In our experience, we find that these components of the algorithm are critical to achieving good performance.

Implementation results

In order to demonstrate the efficacy of our methods, we test them on the domain of 2- and 3-stranded coiled coils (see section 4.4).

First, we show how to use our methods to recognize 3-stranded coiled coils given examples of 2-stranded coiled coils. In other words, starting with a base motif of 2-stranded coiled coils, we learn the target motif comprising of 2- and 3-stranded coiled coils. The initial predictor already has good performance on 2-stranded coiled coils, so we test our algorithm by its performance on 3-stranded coiled coils.

We evaluate our algorithm on 3-stranded coiled coils with respect to two statistical cross validation tests: the "leave one out" test and the "leave half out" test. In the first scenario, the algorithm starts with data from the 2-stranded coiled coil database, and iterates on a test set that contains sequences which are known to form 3-stranded coiled coils, sequences which are thought to form 3-stranded coiled coils, sequences for which no structural information is available, and sequences which are known *not* to contain coiled coils. The category of each sequence in this test set is *not* known to the algorithm, and the sequences which do not contain coiled coils are given to the algorithm in order to test its robustness. At the end of the procedure, the algorithm is evaluated by the number of the 3-stranded coiled coil sequences which it recognizes. Each time a sequence that is present in the database the algorithm is building is scored, it is removed from that database to avoid the possibility of unfairly biasing the test. In this scenario, we find that our algorithm greatly enhances the recognition of 3-stranded coiled coils, without affecting its performance on sequences that are known not to contain coiled coils. In particular, we are able to select 93% of the sequences that are conjectured by biologists to contain coiled coils, with no false positives out of the 286 sequences known not to contain coiled coils. Previously, the best performance without false positives is 67%.

We also test our algorithm on 3-stranded coiled coils in a much more difficult scenario.

In particular, instead of cross validating our procedure by leaving out just one sequence at a time when testing, the algorithm iterates on test sequences that contain only half of the sequences known to form 3-stranded coiled coils. It is then evaluated by its performance on the 3-stranded coiled coil sequences that are not iterated upon. In this scenario, we also find improved performance. The 3-stranded coiled coil sequences are split in half 3 times, and on average, the algorithm is able to select 85% of the left out 3-stranded coiled coil sequences, with likelihood scores higher than that of the highest scoring negative sequence. On average, the previous best performance without false positives is 67%.

Finally, we test our program on subfamilies of 2-stranded coiled coils using the leave one out criterion. For 2-stranded coiled coils, we have a good data set consisting of a diverse set of sequences. However, to test our program, we simulate a limited data problem by testing our program LearnCoil on subfamilies of 2-stranded coiled coils. That is, one subfamily of 2-stranded coiled coils is chosen to make up the base motif, and the class of all 2-stranded coiled coils is the target motif. Here we find that we have excellent performance; i.e., we are able to completely learn the coiled coil regions in our entire 2-stranded coiled coil database starting from a database consisting of coiled coils from any one subfamily. Based on our experiments, such performance does not appear to be possible without the use of our iterative algorithm. In particular, the best performance for the non-iterative approach ranges between 70 and 88%.

Biological significance

As a consequence of this work, we have identified many new sequences that we believe contain coiled coils or coiled-coil-like structures, such as the envelope proteins of mouse hepatitis virus and human rotavirus. One of our more striking findings is the existence of one and occasionally two coiled-coil-like regions in the envelope proteins of many retroviruses, including Human Immunodeficiency Virus (HIV), Simian Immunodeficiency Virus (SIV), and Human T-cell Lymphotropic Virus (HTLV). Independent experimental investigations have also predicted these coiled-coil-like regions in HIV and SIV [19, 56].

4.2 Further background

The coiled coil motif is found in fibrous proteins, DNA binding proteins, and in tRNA-synthetase proteins. Recently it has been proposed that the 3-stranded coiled coil motif acts as the cell fusion mechanism for many viruses, and algorithms for predicting these structures could aid in the study of how viruses invade cells. Computational methods [14, 58] have already identified such coiled coil regions in influenza virus hemagluttinin and Moloney murine leukemia virus envelope protein; both of these predictions have been corroborated in the laboratory [30, 40].

Coiled coils are a particular type of α-helix, consisting of two or more α-helices wrapped around each other with a slight left-handed superhelical twist. Coiled coils have a cyclic repeat of seven positions, a, b, c, d, e, f, and g (see Figure 1). The seven positions are spread out along two turns of the helix. Coiled coils show a characteristic heptad repeat with hydrophobic residues found in positions a and d, and this repeat makes coiled coils particularly amenable to recognition by computational techniques.

Computational methods have been quite successful for predicting coiled coils [63, 58, 42, 12, 13, 14]. These techniques can be described, broadly, as follows:

1. Collect a database of known coiled coils and available amino acid subsequences.
2. Determine whether the unknown sequence shares enough distinguishing features with the known coiled coils to be considered a coiled coil.

Standard approaches [63, 58] look at the frequencies of each amino acid residue in each of the seven repeated positions. Overall this *singles method* does pretty well. When the NewCoil program of Lupas et al. [58] is tested on the PDB (the database of all solved protein structures), it finds all sequences which contain coiled coils. On the other hand, 2/3 of the sequences it predicts to contain coiled coils do not. That is, the false positive rate for the standard method is quite high.

These approaches based on the singles method build a table from the coiled coil database that represents the relative frequency of each amino acid in each position; that is, there is a table entry for each amino acid/coiled coil position pairing. For example, for Leucine and position a, the entry in the table is the percentage of position a's in the coiled coil database which are Leucine, divided by the percentage of residues in Genbank (a large protein sequence database)

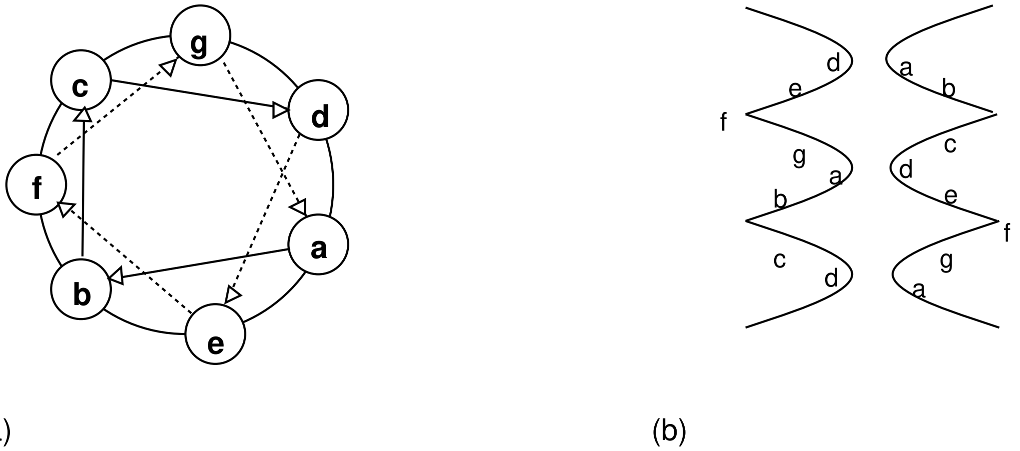


Figure 4.1: (a) Top view of a single strand of a coiled coil. Each of the seven positions $\{a, b, c, d, e, f, g\}$ corresponds to the location of an amino acid residue which makes up the coiled coil. The arrows between the seven positions indicate the relative locations of adjacent residues in an amino acid subsequence. The solid arrows are between positions in the top turn of the helix, and the dashed arrows are between positions in the next turn of the helix. (b) Side view of a 2-stranded coiled coil. The two coils are next to each other in space, with the a position of one next to the d position of another. The coils also slightly wrap around each other (not shown here).

which are Leucine. For example, if the percentage of position a's in the coiled coil database which are Leucine is 27%, and the percentage of residues in Genbank which are Leucine is 9%, then the table entry value for the pair Leucine and position a is 3. Intuitively, this table entry represents the "propensity" that Leucine is in position a in a coiled coil.

The singles method approach [58] actually looks at 28−long windows, since stable coiled coils are believed to be at least 28 residues long. Thus for each residue, it looks at each possible position (a through g), and at all 28-long windows that contain it. It then calculates the relative frequencies for each residue in the window. If the product of the relative frequencies for each residue in some window is greater than some threshold, it concludes that the residue is part of a coiled coil.

Recently researchers have put this problem within a probabilistic framework [12, 13, 14], and have given linear-time algorithms for predicting coiled coils by approximating dependencies between positions in the coiled coil using pairwise frequencies. This method for prediction uses estimates of probabilities for singles and pair positions. For example, in addition to estimating the probability that a Leucine appears in position a of a coiled coil, it also estimates the probability that a Leucine appears in position a of a coiled coil with a Valine appearing in the following d position. For a given residue's contribution to the score, the algorithm considers residues at the structurally relevant distances $i = 1$, $i = 2$ and $i = 4$, calculating the geometric mean of the three quantities $P(k, k+i)/P(k+i)$, where $P(k, k+i)$ is the probability of finding residues k and $k+i$ distance i apart in a coiled coil, and $P(k+i)$ is the probability of finding residue $k+i$ in a coiled coil.

This method of predicting coiled coils has been very effective. When tested on the PDB, the PairCoil algorithm based on this method selects out all sequences that contain coiled coils, and rejects all the sequences that do not contain coiled coils. Furthermore, when tested on a database of 2-stranded coiled coils (with a sequence removed from the database at the time it is scored), each amino acid residue in a coiled coil region is correctly labeled as being part of a coiled coil.

Since the PairCoil algorithm has better performance than the singles method algorithm, particularly with respect to the false-positive rate, this is the scoring method we build on, as well as the scoring method to which we compare our results.

Other types of iterative approaches have been applied to sequence alignment and protein structure prediction by researchers [73, 3, 46, 36]. Algorithmically, our approach differs from these approaches in two major ways. The first is our use of randomness to incorporate sequences into our database, and the second is our use of weighting to update the database (see section 4.3). In addition, several of these papers are directed toward sequence alignment, and sequence alignment is not so effective a tool for predicting coiled coils, as the various subfamilies of coiled coils do not align well to each other. Also, since the goal of these other methods is often to output potential matching alignments, the testing of these algorithms is quite different. In particular, although some of these approaches use the "leave one out" criterion, to the best of our knowledge, none of them test performance with the "leave half out" criterion.

Various machine learning techniques have been applied to the protein structure prediction problem. The two main approaches are neural nets (e.g., [47, 67, 59]) and hidden Markov models (e.g., [53, 9]). Both of these approaches require adequate data on the target motif, since there is a "training session" on sequences that are *known* to contain the target motif. Our approach differs from these methods since it does not require well analyzed data on the

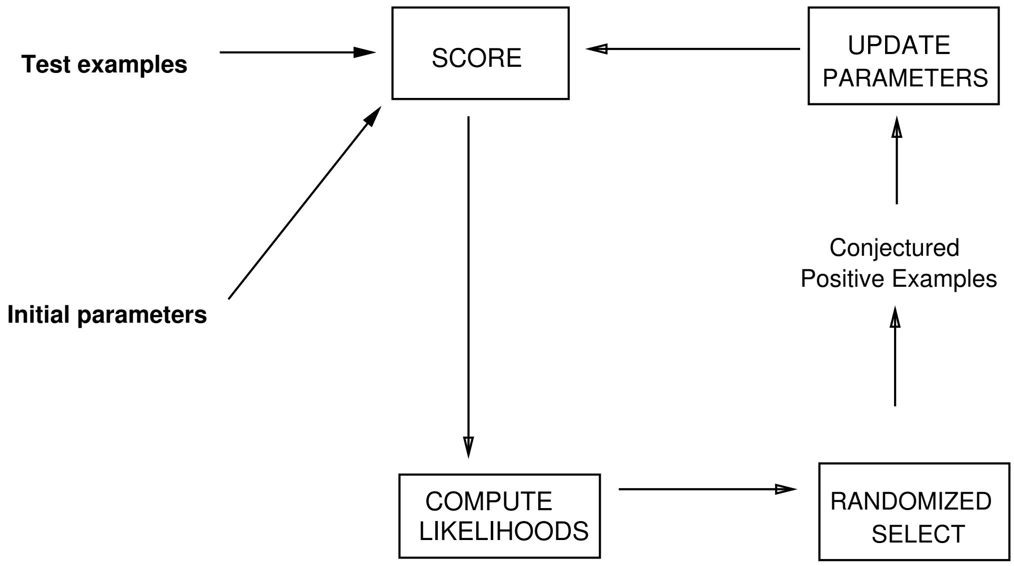


Figure 4.2: Our basic learning algorithm. Initially, the algorithm starts off with a test set of examples and a set of initial parameters. In each iteration, the algorithm selects new examples, and re-estimates its parameters.

target motif per se. Instead it uses already available data on a base motif and generalizes it to recognize the target motif, by running on a large number of sequences, some of which are suspected to fold into the target motif.

Other learning approaches which have been applied to protein structure prediction include rule-based methods (e.g., [60]).

4.3 The algorithm

We first describe the general framework for our algorithm. Namely, we are initially given a set of parameters that help characterize our *base concept*, and a set of test examples. Our goal is to decide which of these test examples are positive examples of some *target concept*. In addition, we know that the target concept is a generalization of the base concept. Our algorithm takes advantage of the fact that the base concept is somewhat related to the target concept. In particular, once the algorithm has identified some of the test examples that are presumed to be related to the base concept, it can modify its database by "adding" these newly found examples. Examples are selected by a randomized procedure based on likelihoods. This process is then iterated, as the added examples change the scores of other examples. (See figure 4.2.)

We have implemented our learning algorithm for the protein motif recognition problem. In particular, our learning algorithm LearnCoil proceeds as follows. It is given two inputs: a database of a base motif which is related to the target motif we are interested in, and a large database of *iteration test sequences* which is comprised of sequences that we believe contain the target motif as well as many other sequences of unknown structure. In practice, we generally include in the iteration test sequences some fraction of the PIR (a large protein sequence database), the sequences from the PDB (the database of solved protein structures) that are known not to fold into the target motif, and sequences conjectured by biologists to fold into the target motif.

Initially, the algorithm estimates pair and singles amino acid residue probabilities for the motif's positions. Then the algorithm iterates four basic steps:

1. The algorithm uses its estimates of the pair and singles probabilities to determine a likelihood function, which maps residue scores to a likelihood of the residue belonging to the target motif.

2. The algorithm scores each of the iteration test sequences using the estimated probabilities, and calculates the likelihoods for each of these sequences.

3. The algorithm flips coins with probability proportional to the likelihood of each score to determine which parts (if any) of each sequence are presumed to be part of the target motif. The residues which are thus determined to be presumed examples of the target motif make up the new database for the next iteration.

4. The algorithm uses the base motif database and the new database just determined in this iteration to update its estimates of the singles and pair probabilities for the target motif using a Bayesian-like weighting scheme (see section 4.3.4).

The algorithm continues iterating until the new database stabilizes.

We now describe each of the components of the algorithm in more detail, using coiled coils as an example, although the algorithm can be applied to other protein motifs.

4.3.1 Scoring

In our implementation, we use the PairCoil program described by Berger et al. [14] as our scoring procedure, although any good prediction algorithm with a low false positive rate can be used for scoring. This scoring method uses correlation methods that incorporate pairwise dependencies between amino acids at multiple distances. The scoring procedure gives a *residue score* for each amino acid in a given sequence, as well as a *sequence score*, which is the maximum residue score in the sequence.

In order to use this scoring procedure, we must have estimates for the probabilities for the singles and pair positions for the motif. Initially, we have estimates for the probabilities based on the database of sequences of the base motif, and after each iteration of the algorithm, we use updated probabilities. In each iteration after the first, when we score a sequence we check to see if it was identified in the previous iteration. If it was, we remove this sequence from the database and adjust the probabilities before scoring.

Given good estimates for the probabilities for the singles and pair positions for the motif, and reasonable assumptions about dependencies in the motif, the PairCoil scoring method which we use as a subroutine is mathematically justified [12].

4.3.2 Computing likelihoods

Once we have a sequence score, we assess it by converting it into a likelihood that the sequence contains the target motif. In each iteration of the algorithm, we compute a function that takes a residue score and computes the likelihood that the residue is part of the target motif.

We compute this likelihood function in a manner described in [14]. In particular, every sequence in a large sequence database is scored. (Ideally, this large sequence database is the PIR. However, in practice, to save time, we use a sampled version of the PIR, which is 1/25-th the size; the likelihood function calculated using this sampled PIR is a good approximation to the likelihood function calculated using the entire PIR.) The sampled PIR residue score histograms are nearly Gaussian distributed with some extra probability mass added on the right-hand tail. This extra mass is attributed to residues in the target motif, since they are expected to score higher. In the case of the coiled coil motif, given the biological data currently available, it is estimated that between 1/50 and 1/30 of residues in the PIR are in a coiled coil.

To fit a Gaussian to the histogram data, we calculate the mean so that the extra probability mass on the right side of the mean corresponds to between 1/50 and 1/30 of the total mass of the PIR. We then compute the standard deviation using only scores below that mean, where a Gaussian better fits the histogram data. The likelihood that a residue with a given score is a coiled coil is estimated as the ratio of the extra histogram mass above the Gaussian at that score (corresponding to data assumed to be coiled) to the total histogram mass at that score. A least square fit line is then used to approximate the likelihood function in the linear region from 10 to 90 percent. This line then gives an approximation for the likelihoods corresponding to all scores.

One feature of this method of computing likelihoods is that it does not allow too many residues to be considered as part of coiled coils. This helps keep the false positive rate of the algorithm low.

4.3.3 Randomized selection of the new database

Once we have obtained the likelihood function for an iteration, we wish to use the likelihoods to build a new database of sequences presumed to fold into the target motif. At the beginning of each iteration, our new database contains no sequences. Then for each sequence in the set of test sequences, we do the following. First, we score each sequence and then convert its sequence score to a likelihood. Next, we draw a number uniformly at random from the interval $[0, 1]$. If the number drawn is less than or equal to the likelihood of the sequence, then the sequence is added to the new database. All residues in this sequence that have scores equal to the sequence score or greater than the 50% likelihood score (which is the algorithm's cutoff for a residue being in a coiled coil) are added to the database. Once we have processed every sequence in our test set, then we have our new database of sequences presumed to fold into the target motif.

In practice, we find that adding randomness substantially improves the performance of our algorithm. In fact, if the procedure is written just to accept sequences that have greater than 50% likelihood, then the algorithm fails to recognize many sequences which are known to contain 3-stranded coiled coils. On the other hand, if the procedure lowers the threshold value for acceptance, then its false positive rate increases.

4.3.4 Updating parameters

Once we have a new database of sequences which are thought to contain the target motif, we need to update the parameters used by the algorithm for scoring. In our case, in each iteration of the algorithm, the scoring procedure needs updates of the estimates of probabilities for singles and pair positions. The most straightforward way to update the probabilities is to use a maximum likelihood estimate from frequency counts from the new database. However, this does not work that well in practice. Instead, we update each probability by taking a weighted average of the probability given by the base motif database and the probability given by the new database.

We now describe a theoretical framework for updating probabilities in this manner in each iteration of our algorithm. The approach we give is motivated by a Bayesian viewpoint [45, 15]. In particular, we think of the probabilities we are trying to estimate as the parameters of a Multinomial distribution, and we use the Dirichlet density to model the prior information we have about these probabilities. In fact, the approach we give is not completely Bayesian, as we will use the seen data to pick the parameters of the prior distribution; this is sometimes called a Bayes/Non-Bayes compromise [45].

We will use frequency counts from our databases to estimate singles and pair probabilities. For simplicity, we focus on the case of updating singles probabilities; updating pair probabilities is analogous.

Initially, we have a database of sequences which fold into a particular base motif. Thus, for each position in the motif, we have a 20-long count vector, one for each of the 20 amino acids. For example, for a given database of known coiled coils, for position a, we know how many times each amino acid appears. In addition, after each iteration of the algorithm, we have a *new* database of sequences that we have selected and which we presume fold into the target motif. This new database also gives us a 20-long count vector for each position in the motif.

We update the probabilities using these frequency count vectors. In particular, we fix a numbering of the amino acids from 1 to 20. Then for each position q in the motif (for coiled coils, $q \in \{a, b, c, d, e, f, g\}$), we have a count vector $\vec{x}^{(q)} = (x_1^{(q)}, x_2^{(q)}, \ldots, x_{20}^{(q)})$, where $x_i^{(q)}$ is the number of times amino acid i appears in position q of the motif in the base motif database. In addition, we have a count vector $\vec{y}^{(q)} = (y_1^{(q)}, y_2^{(q)}, \ldots, y_{20}^{(q)})$, where $y_i^{(q)}$ is the number of times amino acid i appears in position q of the motif in the new database (i.e., the database consisting of the sequences we have picked in this iteration of the algorithm).

Let $\vec{p}^{(q)} = (p_1^{(q)}, p_2^{(q)}, \ldots, p_{20}^{(q)})$ be the actual probabilities for the amino acids appearing in position q of the motif. We assume, for simplicity, that the count vectors for each position are independent of each other. Thus, we focus on updating the probabilities of one position independent of the other positions. For notational convenience, we fix a position and drop the superscript q. We assume that for a fixed position, the count vector is generated at random according to the Multinomial distribution with parameter $\vec{p} = (p_1, p_2, \ldots, p_{20})$. The parameters $p_1, p_2, \ldots, p_{20}$ are the "true" probabilities of seeing the amino acids in the fixed position in the motif we are interested in. These are the parameters we wish to estimate.

In our case, we have very strong *a priori* knowledge about the probabilities. Since we are trying to learn a particular target structural motif from a related base structural motif, we can use the probabilities estimated from the base motif as prior probabilities. In fact, because these structural motifs are related, we expect the updated probabilities for the target motif to be similar to the original probabilities for the base motif.

We model our a priori beliefs by the Dirichlet density. The value of a Dirichlet density $\mathcal{D}(\alpha)$ (with parameter $\vec{\alpha} = (\alpha_1, \alpha_2, \ldots, \alpha_k)$, where $\alpha_i > 0$ and $\alpha_0 = \sum \alpha_i$) at a particular point $\vec{x} = (x_1, x_2, \ldots, x_k)$, where $\sum x_i = 1$ is given by:

$$f(\vec{x}|\vec{\alpha}) = \frac{\Gamma(\alpha_0)}{\prod_{i=1}^{k} \Gamma(\alpha_i)} \prod_{i=1}^{k} x_i^{(\alpha_i - 1)}.$$

The gamma function $\Gamma(\alpha)$ is:

$$\Gamma(\alpha) = \int_0^\infty e^{-x} x^{\alpha-1} dx.$$

The mean of Dirichlet density is $(\alpha_1/\alpha_0, \alpha_2/\alpha_0, \ldots, \alpha_k/\alpha_0)$, and the larger α_0 is, the smaller the variance is.

Thus a Bayesian estimate for the probabilities $p_1, p_2, \ldots, p_{20}$ can be found by looking at the posterior distribution. The Dirichlet distribution is conjugate for the Multinomial, and the posterior distribution is the Dirichlet distribution $\mathcal{D}(\vec{\alpha} + \vec{y})$ [15, 45]. That is, the new parameter of the distribution is the vector sum of the original parameters and the observed data. Thus, a

Bayesian estimate for probability p_i after seeing the data $\vec{y}$ is

$$\frac{\alpha_i + y_i}{\alpha_0 + y_0}, \text{ where } y_0 = \sum_{i=1}^{20} y_i.$$

We still have not addressed the issue of how the parameters of the prior distribution are chosen. We depart from the traditional Bayesian approach, and choose the parameters of the prior distribution after seeing the data. In particular, since the base motif and the target motif are related, we want the base motif database to have a strong effect on the estimates for our probabilities, and thus we choose the variance of the prior distribution accordingly.

The mean of the Dirichlet density is specified by the estimated probabilities of the base motif. The variance of the density is picked as follows. If $0 < \lambda < 1$ is the effect, or weight, that we want the base motif database to have, then we let $\alpha_i = x_i \cdot \frac{\lambda}{1-\lambda}\frac{y_0}{x_0}$, where $x_0 = \sum_{i=1}^{20} x_i$ and $y_0 = \sum_{i=1}^{20} y_i$. (Actually, we have to be careful in the case where $x_i = 0$.) It is easy to verify that our estimate for the probability p_i is given by $\lambda\frac{x_i}{x_0} + (1-\lambda)\frac{y_i}{y_0}$. Namely, our updated probability is a weighted average of the probability given by the base motif database and the probability given by the new database.

In practice, we have found that our method of updating probabilities has worked well. It is superior to a maximum likelihood approach which uses just the current iteration's frequency counts. These estimates of the probabilities are especially problematic in the zero frequency case. Our method also performs better than an unweighted approach using both the initial frequency counts and the current iteration's frequency counts. These estimates of the probabilities are largely dependent on the size of the original database, and the number of residues that are presumed at each iteration to be part of the target motif. In our test domain of coiled coils, we found that this method of updating probabilities missed more sequences that contain coiled coils than did our method for updating probabilities.

Using Dirichlet mixture densities as priors to estimate amino acid probabilities has been studied by Brown et al. [29]. Their approach uses as a prior the maximum likelihood estimate of a mixture Dirichlet density, based on data previously obtained from multiple alignments of various sets of sequences. Their approach is a pure Bayesian approach, and their prior distribution has a smaller effect on the final probability estimates.

4.3.5 Algorithm termination

The iteration process terminates when it stabilizes; that is, when the number of residues added from the previous iteration changes by less than 5%. Usually the procedure converges in around six iterations; otherwise, we terminate it after 15 iterations. In practice, we found that the algorithm rarely had to to be terminated due to lack of convergence.

In our implementation, the running time of the entire algorithm is linear in the total number of residues in all sequences which are given as input. The basic operation in each iteration is scoring every sequence using the PairCoil algorithm. For each sequence, the PairCoil scoring program takes time linear in the number of residues. Since we have at most a fixed number of iterations, the entire algorithm is linear-time.

After running LearnCoil, the “learned” target concept contains both 2- and 3-stranded coiled coils. The problem of distinguishing one set from the other remains. The MultiCoil program of Wolf, Kim, and Berger [unpublished results] is being developed for this purpose and in initial experiments performs well.

4.4 Results

We have implemented our algorithm in a C program called LearnCoil. We test our program on the domain of 3-stranded coiled coils and subclasses of 2-stranded coiled coils. First we describe the databases we use to test the program, and then we follow by describing the program's performance.

4.4.1 The databases and test sequences

Our original database of 2-stranded coiled coils consists of $58,217$ amino acid residues which were gathered from sequences of myosin, tropomyosin, and intermediate filament proteins [14]. We also have separate databases containing sequences from each of these protein subclasses individually. A synthetic peptide of tropomyosin is the only solved structure among these.

We test LearnCoil on the 3-stranded coiled coils by starting the algorithm with the base database of all 2-stranded coiled coils. We test LearnCoil on the 2-stranded coiled coils by starting the algorithm with a base database of one of the subfamilies of the 2-stranded coiled coils.

The set of iteration test sequences for testing performance on 3-stranded coiled coils consists of the following 5516 sequences: 286 known non-coiled coils from the non-redundant version of the PDB created in [14] (the PDB is the database of solved protein structures); 2% of the sequences in OWL (a large non-redundant composite database, where no two sequences in the database are exactly the same and no two sequences show only "trivial" differences [20]), with any obvious members of the PDB removed (2815 total); sequences in OWL whose names contain the strings actinin, alpha spectrin, dystrophin, tail fiber, laminin, fibrinogen, env, spike, glycoprotein, bacteriophage T4 wac, bacteriophage K3 fibritin, heat shock transcription, or macrophage scavenger receptor, as well as the 3-stranded coiled coil mutant for GCN4 (2415 total, of which many are thought to contain 3-stranded coiled coils, and the 46 sequences given below are known to contain them).

The 3-stranded coiled coil set is comprised primarily of laminin and fibrinogen sequences, as well as influenza virus hemagluttinin, Moloney murine leukemia envelope protein, 2 heat shock transcription factors, bacteriophage T4 and K3 wac proteins, the trimeric GCN4 mutant, 2 macrophage scavenger receptors, and bacteriophage T3 and T7 tail fibers.

Our set of iteration test sequences for 2-stranded coiled coils includes: 1/23 of the PIR (1553 total); the 286 known non-coiled coils; and the two of the subfamilies out of myosins, tropomyosins, and intermediate filaments. (For example, when we start with a database of intermediate filaments, our iteration test sequences include myosins and tropomyosins.)

Note that most of the sequences in our 2- and 3-stranded coiled coil data sets do not have solved structures. However, there is strong experimental support that they contain coiled coils, although often the boundaries of the coiled coil regions are difficult to specify exactly. We do not know the three dimensional structure for most of the protein sequences in our iteration test sets (except for the sequences from the PDB and portions of the sequences making up the 2- and 3-stranded coiled coil data sets).

4.4.2 Learning 3-stranded coiled coils

Our techniques improve non-learning based approaches, such as PairCoil [14], which often fails to identify 3-stranded coiled coil regions.

Base Set	Evaluation Set	Performance without LEARNCOIL		Performance with LEARNCOIL	
		% of seqs	# of false positive seqs	% of seqs	# of false positive seqs
2-str CCs	46 3-str CCs	67%	0/286	93%	0/286

Table 4.1: Learning 3-stranded coiled coils from 2-stranded coiled coils using the leave one out criterion.

We test the algorithm on 3-stranded coiled coils in two ways: the "leave one out" test and the "leave half out" test. In both cases, LEARNCOIL improves recognition of 3-stranded coiled coils starting with an initial database of 2-stranded coiled coils. We measure LEARNCOIL's performance on the 286 non-coiled coil proteins, and an evaluation set consisting of 3-stranded coiled coil sequences. We assume that a false negative prediction has occurred when a sequence in the 3-stranded coiled coil evaluation set receives a score with a corresponding likelihood less than 50%. We assume a false positive has occurred when a non-coiled coil protein scores at least 50% likelihood. Since our algorithm is randomized, the final likelihoods are found by averaging LEARNCOIL outputs over five runs.

In the first "leave one out" scenario, the algorithm is run with all the 5516 iteration test sequences described in section 4.4.1. Once the algorithm terminates, each of the 46 sequences in the 3-stranded coiled coil set is scored with respect to parameters calculated from the new database in the final iteration minus the effects of this sequence. That is, since the 46 3-stranded coiled coil sequences are included in the iteration test set, if a sequence appears in the final database, before scoring this sequence, the sequence is removed to avoid the possibility of unfairly biasing the test.

The weight of the original database (i.e., relative to the new database) was chosen empirically to be $\lambda = 0.1$. This makes sense because 2- and 3-stranded coiled coils are sufficiently different; thus, it may require much more weight for the newly identified sequences to effectively broaden the new database to contain 3-stranded coiled coils. We also experimented with weights in the range $0 \leq \lambda \leq 0.5$ but $\lambda = 0.1$ gave the best results.

Our algorithm LEARNCOIL positively identifies 43 out of 46 (93%) of the 3-stranded coiled coil sequences and makes no false positive predictions. In contrast, PAIRCOIL positively identifies 31 out of 46 (67%) of the 3-stranded coiled coils and also makes no false positive predictions (see Table 4.1). Moreover, using the final databases that LearnCoil produced, we are able to recognize all the sequences in the 2-stranded coiled coil database. Thus the final databases produced by the LearnCoil algorithm performs well on both 2- and 3-stranded coiled coils.

In the second "leave half out" scenario, we split the 3-stranded coiled coil sequence set in half in the following manner. First, the 46 3-stranded coiled coil sequences are divided into the following subgroups: α-fibrinogens, β-fibrinogens, γ-fibrinogens, laminins, tail fibers, heat shocks, and all remaining protein sequences. Next, each of these subgroups is randomly divided into two parts, one for each half; this ensures that in the final split, each half is fairly representative of examples of the 3-stranded coiled coil motif.

We split the 3-stranded coiled coil sequences 3 times in the above manner. This then gives us six different iteration and evaluation sets. Each evaluation set consists of 23 3-stranded coiled coil sequences, and the corresponding iteration test set consists of 5493 sequences (the original 5516 sequences, minus the 23 sequences in the evaluation set). We run LearnCoil on each of the six iteration test sets, and evaluate the algorithm by its performance on the corresponding evaluation sets (namely, those 3-stranded coiled coil sequences which are not included in the iteration test set). Note that the set of sequences with solved structures that do not contain coiled coils are included in all iteration test sets, and are scored using the leave one out criterion.

For each iteration test set, our algorithm is again run 5 times with $\lambda = .1$, and with final likelihoods averaged over the runs. Table 4.2 gives the performance of our algorithm on the different evaluation sets. On average, LearnCoil selects out 85% of the 3-stranded coiled coil sequences not originally in the set of sequences upon which it iterates. In contrast, PairCoil on average selects out 67% on the same sets of sequences. In all but one of the six experiments, the algorithm does not get any false positives from the set of solved structures. In the one scenario when it does get a false positive, the likelihood of all sequences in the corresponding evaluation set (B1) that score above 50% also score higher than this false positive.

The average performance of LearnCoil on the 3-stranded coiled coil sequences included in the iteration test set is 88%. (Individual performance data for each of the six experiments is not shown.) This average does not seem to be significantly higher than the algorithm's average

Base Set	Evaluation Set	Performance without LEARNCOIL		Performance with LEARNCOIL	
		% of seqs	# of false positive seqs	% of seqs	# of false positive seqs
2-str CCs	Set A1, 23 3-str CCs	65%	0/286	87%	0/286
2-str CCs	Set A2, 23 3-str CCs	70%	0/286	83%	0/286
2-str CCs	Set B1, 23 3-str CCs	74%	0/286	87%	1/286
2-str CCs	Set B2, 23 3-str CCs	61%	0/286	78%	0/286
2-str CCs	Set C1, 23 3-str CCs	70%	0/286	96%	0/286
2-str CCs	Set C2, 23 3-str CCs	65%	0/286	78%	0/286

Table 4.2: Learning 3-stranded coiled coils from 2-stranded coiled coils using the leave half out criterion. The 3-stranded coiled coil sequences are split 3 times, giving us six different iteration and evaluation sets. The evaluation sets are A1, A2, B1, B2, C1 and C2 (A1 and A2 are a result of one split, etc.).

performance on the sequences in the evaluation set. Thus in comparing the results in Table 4.2 with the results in Table 4.1, it appears that the decreased performance on these runs with the splits is the result of fewer available 3-stranded coiled coil sequences to the algorithm, and not upon whether the evaluation criterion is the leave one out criterion or the leave half out criterion.

4.4.3 Learning subclasses of 2-stranded coiled coils

Our results on subclasses of the 2-stranded coiled coil motif indicate that we are able to "learn" coiled coil regions in one family of proteins using a database consisting of coiled coils from another family of proteins. For example, we are able to learn coiled coils in intermediate filaments from a database of coiled coils in either myosins or tropomyosins. Our techniques improve non-learning based approaches, such as the PAIRCOIL program [14], which fail to identify conjectured coiled coil residue positions.

We tested LEARNCOIL on three different domains (Table 4.3): tropomyosins (TROPs) as a base set and myosins (MYOs) and intermediate filaments (IFs) as an evaluation set; myosins as a base set and tropomyosins and IFs as an evaluation set; IFs as a base set and myosins and tropomyosins as an evaluation set. A different set of iteration test sequences was used for

Base Set	Evaluation Set	Performance without LearnCoil		Performance with LearnCoil	
		% of residues	# of false positive seqs	% of residues	# of false positive seqs
TROPs	MYOs + IFs	71%	4/286	99%	1/286
MYOs	TROPs + IFs	89%	2/286	99%	1/286
IFs	MYOs + TROPs	83%	4/286	99%	2/286

Table 4.3: Learning 2-stranded coiled coils from a restricted set

each of these tests; that is, the set that includes sequences of the two protein families in the evaluation set. For these experiments, we have residue data, and thus our performance measure is with respect to these. False negatives are residues of sequences in the evaluation set which do not have at least a 50% likelihood. False positives are defined as in section 4.4.2

Here the weight of the original database was empirically chosen to be $\lambda = 0.3$. One possible explanation for this is since the subclasses of 2-stranded coiled coils has more similarities than differences, the program does not have to be so aggressive in picking up the evaluation set. Moreover, the goal is a target set of 2-stranded coiled coils, and this is best achieved by weighting each of the 3 types of proteins equally. We also experimented with weights of $\lambda = 0.1$ and $\lambda = 0.5$, and while their overall performance was similar, they produced more false positives.

First, we consider experiments with tropomyosins in the base set and myosins and IFs in the evaluation set. LearnCoil positively identifies 99% of the myosin and IF residues in the 2-stranded database and makes one false positive prediction. This is in contrast to PairCoil, which obtained a performance of 70.9%, with four false positive and two false negative predictions.

Next we consider experiments with a base set of myosins and an evaluation set of tropomyosins and IFs. LearnCoil positively identifies 99% of the tropomyosin and IF residues and makes one false positive prediction. This is in contrast to PairCoil, which obtained a performance of 88.8%, with two false positive and one false negative predictions.

Lastly, we consider experiments with a base set of IFs and an evaluation set of tropomyosins and myosins. LearnCoil positively identifies 99.4% of the tropomyosin and IF residues and makes two false positive predictions. One possible explanation for more false positives here is that the IFs have a less obvious coiled-coil structure and there very well may be non-coiled coil residues in the database; consequently, starting with a table of solely IFs may select out non-coiled coils for the target database. In contrast, PairCoil obtained a performance of 83.3%, with four false positive predictions.

For all three above experiments, LearnCoil improved performance of PairCoil in identifying coiled coil residues, while also improving its false positive rate.

We also tested LearnCoil with the NewCoils program [58] used as the underlying scoring algorithm. For subclasses of 2-stranded coiled coils, we found that LearnCoil enhanced the performance of NewCoils as well. It obtained a performance of 96.2% when tropomyosins were used as the base set, a performance of 95.3% when myosins were used, and a performance of 98.2% when IFs were used. The program did not make any false positive predictions when run on these three test domains. In contrast, the non-learning based version of NewCoils had substantial overlap between the residue scores for coiled coils and non-coiled coils in all of the three test domains.

4.4.4 New coiled-coil-like candidates

The LearnCoil program has identified many new sequences that we believe contain coiled-coil-like structures. Table 4.4 lists some examples of "newly found" viral proteins (i.e., proteins for which PairCoil indicates that no coiled coil is present, but LearnCoil indicates a coiled-coil-like structure is present). We believe that the proteins given in Table 4.4 either contain coiled coils or coiled-coil-like structures. For example, recent biological work has identified a coiled-coil-like structure which is believed to consist of a parallel, trimeric coiled coil encircled by three helices packed in an antiparallel formation; this structure is thought to be in the envelope glycoproteins of both HIV and SIV (Simian Immunodeficiency Virus) [19, 56].

Our program seems to be able to accurately predict this new coiled-coil-like structure. For example, it identifies two coiled-coil-like regions in the envelope protein of SIV. Independently, the biological investigation of SIV by Blacklow et al. predicts that these are the two regions that are part of the coiled-coil-like structure [19]. One of these regions (comprising the outer three helices) is predicted by the NewCoil program and is given a 26% likelihood by the PairCoil program. The other region (comprising the trimeric coiled coil) is only predicted by our LearnCoil program. This region corresponds to the N-terminal fragment in the paper of Blacklow et al. In fact, the region LearnCoil predicts and the region that Blacklow et al. find are almost identical: LearnCoil predicts a coiled-coil-like structure starting at residue 553 and ending at residue 601, whereas Blacklow et al. start the region at residue 552 and end it at residue 604.

PIR Name	LearnCoil Likelihood	PairCoil Likelihood
mouse hepatitis virus E2 glycoprotein precursor	>90%	23%
human rotavirus A glycoprotein NCVP5	>90%	<10%
human respiratory syncytial virus fusion glycoprotein	>90%	<10%
human T-cell surface glycoprotein CD4 precursor	77%	<10%
human T-cell lymphotropic virus – type I, env	>90%	<10%
equine infectious anemia virus, env	>90%	<10%
fruit fly 14-3-3 protein	52%	<10%
HIV, env	>90%	<10%
SIV, env	>90%	26%

Table 4.4: Newly discovered coiled-coil-like candidates

Moreover, there is biological evidence that several other of the sequences in Table 4.4 contain coiled-coil-like structures. Our predictions were made independently of these results. Recently, the crystal structure of two 14-3-3 proteins have been solved [55, 75]. The paper of Liu et al. studies the zeta transform of the 14-3-3 structure in E. coli, and they report a 2-stranded anti-parallel coiled coil structure. On the other hand, the paper of Xiao et al. studies the human T-cell τ dimer, and they report helical bundles. Although there is some uncertainty here, it is likely that the 14-3-3 protein we have identified contains a coiled-coil-like structure, if not a coiled coil itself. The Human T-cell lymphotropic virus and equine infectious anemia virus are closely related to HIV, and thus their envelope proteins are also likely to contain coiled-coil-like structures.

The proteins reported in Table 4.4 are compared to the PairCoil program. The NewCoil program of Lupas et al. finds some of these proteins; however, in general, this program finds a significant number of false positives. The 14-3-3 protein, the human T-cell lymphotropic virus envelope protein and the human T-cell surface glycoprotein CD4 precursor are found only using our LearnCoil program. As mentioned above, there is some biological evidence that at least two of these proteins (the 14-3-3 protein and human T-cell lymphotropic virus envelope protein) contain coiled-coil-like structures.

We anticipate that the identification of likely coiled-coil-like regions in important protein sequences (such as those in Table 4.4) will facilitate and expedite the study of protein structure by biologists. In addition, since our program LearnCoil is able to identify the new coiled-coil-like motif in HIV and SIV, it is possible that our program will help aid in the discovery of this structure in other retroviruses.

4.5 Conclusions

In this chapter, we have shown that a learning-based algorithm that uses randomness and statistical techniques can substantially enhance existing methods for protein motif recognition. We have designed a program LearnCoil and demonstrated its ability to "learn" the 2-stranded and 3-stranded coiled coil motif. It has identified new sequences that we believe contain coiled-coil-like structures. It is our hope that biologists will use this program to help identify other new coiled-coil-like structures.

There is evidence that our program may have identified a new coiled-coil-like motif that occurs in retroviruses, and future work involves studying retroviruses and this motif more closely.

In the future we plan to apply the LearnCoil program to motifs other than those that have coiled-coil-like properties. Limited data is a problem for many protein structure prediction problems. There are newly discovered protein motifs for which biologists cannot yet predict, and more importantly, do not yet even know the structural features that characterize the motifs. We hope to extend the techniques developed here to aid in the determination of crucial structural features that give rise to these motifs, as well as to learn how to predict which proteins exhibit this motif.

Chapter 5

Concluding remarks

In this thesis, we have studied three problems in machine learning. In the first part of the thesis, we examined Valiant's PAC model, and considered learnability in this model. In particular, we studied concept classes of functions on k terms, and gave an algorithm for learning any function on k terms by general DNF. On the other hand, we showed that if the learner is restricted so that it must output a hypothesis which is a member of the concept class being learned, then learning the concept class of any symmetric function on k terms is NP-hard (except for the concept classes of AND, NOT AND, TRUE and FALSE). Our results completely characterize the learnability of concept classes of symmetric functions on k terms. We leave as an open problem whether concept classes for more general functions on k terms can be learned when the learner's output hypothesis is restricted.

The second part of the thesis introduced the problem of piecemeal learning an unknown environment. For environments that can be modeled as grid graphs with rectangular obstacles, we gave two piecemeal learning algorithms in which the robot traverses a linear number of edges. For more general environments that can be modeled as arbitrary undirected graphs, we gave a nearly linear algorithm. An interesting open problem is whether there exists a linear algorithm for piecemeal learning arbitrary undirected graphs. Piecemeal learning takes into account just one of the limitations on a robot's resources. It would be interesting to come up with models and algorithms to handle other practical limitations of a robot, such as incorrect data that a robot may receive (due to noisy sensors) and difficulties a robot may have in motor control. Other extensions of the work might include the scenario of multiple robots, or multiple "refueling stations."

In the last part of the thesis, we applied machine learning techniques to the problem of protein folding prediction. We gave an iterative learning algorithm that is particularly effective for folds for which there is not much currently available data. We implemented our algorithm, and showed its effectiveness on the 3-stranded coiled coil motif. There are other motifs for which there is a lack of data, such as β-rolls and β-helices, and it would interesting to extend our techniques to work on these motifs. In addition, there is evidence that our program may have identified a new coiled-coil-like motif that occurs in retroviruses, and future work involves studying this motif more closely.

Bibliography

[1] Dana Angluin. On the complexity of minimum inference of regular sets. *Information and Computation*, 39:337–350, 1987.

[2] Dana Angluin. Computational learning theory: Survey and selected bibliography. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, pages 351–369, May 1992.

[3] T. K. Attwood and J. B. C. Findlay. Design of a discriminating fingerprint for G-protein-coupled receptors. *Protein Engineering*, 6:167–176, 1993.

[4] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear cost constructions of neighborhood covers in sequential and distributed environments and their applications. In *Proceedings of the Thirty-Fourth Annual Foundations of Computer Science*, pages 638–647, November 1993.

[5] Baruch Awerbuch, Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal graph exploration by a mobile robot. In *Proceedings of the Eighth Conference on Computational Learning Theory*, Santa Cruz, CA, July 1995.

[6] Baruch Awerbuch and Robert G. Gallager. Distributed BFS algorithms. *The 26th Symposium on Foundations of Computer Science*, pages 250–256, October 1985.

[7] Baruch Awerbuch and Robert G. Gallager. A new distributed algorithm to find breadth first search trees. *IEEE Transactions on Information Theory*, IT-33(3):315–322, 1987.

[8] Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J. E. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, October 1993.

[9] P. Baldi. Hidden Markov models in molecular biology. Technical report, JPL California Institute of Technology, 1993.

[10] E. Bar-Eli, P. Berman, A. Fiat, and P. Yan. On-line navigation in a room. In *Symposium on Discrete Algorithms*, pages 237–249. SIAM, 1992.

[11] Michael A. Bender and Donna K. Slonim. The power of team exploration: two robots can learn unlabeled directed graphs. In *Proceedings of the Thirty-Fifth Annual Symposium on Foundations of Computer Science*, pages 75–85, November 1994.

[12] B. Berger. Algorithms for protein structural motif recognition. *Journal of Computational Biology*, 2:125–138, 1995.

[13] B. Berger and D. Wilson. Improved algorithms for protein motif recognition. In *Symposium on Discrete Algorithms*, pages 58–67. SIAM, January 1995.

[14] B. Berger, D. B. Wilson, E. Wolf, T. Tonchev, M. Milla, and P. S. Kim. Predicting coiled coils using pairwise residue correlations. *Proceedings of the National Academy of Sciences*, 92:8259–8263, 1995.

[15] J. Berger. *Statistical Decision Theory and Bayesian Analysis*. Springer-Verlag, New York, 1985.

[16] Margrit Betke. Algorithms for exploring an unknown graph. Master's thesis, MIT Department of Electrical Engineering and Computer Science, February 1992. (Published as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-536, March, 1992).

[17] Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal learning of an unknown environment. In *Proceedings of the Sixth Conference on Computational Learning Theory*, pages 277–286, Santa Cruz, CA, July 1993. (Published as MIT AI-Memo 1474 and CBCL-Memo 93.).

[18] Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal learning of an unknown environment. *Machine Learning*, 18(2/3):231–254, March 1995.

[19] S. Blacklow, M. Lu, and P. S. Kim. A trimeric subdomain of the Simian Immunodeficiency Virus envelope glycoprotein. *Biochemistry*, 34:14955, 1995.

[20] A. J. Bleasby and J.C. Wooton. *Protein Engineering*, 4, 1990.

[21] Avrim Blum and Prasad Chalasani. An on-line algorithm for improving performance in navigation. In *Proceedings of the Thirty-Fourth Annual Symposium on Foundations of Computer Science*, pages 2–11, November 1993.

[22] Avrim Blum, Prabhakar Raghavan, and Baruch Schieber. Navigating in unfamiliar geometric terrain. In *Proceedings of Twenty-Third ACM Symposium on Theory of Computing*, pages 494–504. ACM, 1991.

[23] Avrim Blum and Mona Singh. Learning functions of k terms. In *Proceedings of the Third Annual Workshop on Computational Learning Theory*, pages 144–153. Morgan Kaufmann, 1990.

[24] Manuel Blum and Dexter Kozen. On the power of the compass (or, why mazes are easier to search than graphs. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. IEEE, 1978.

[25] Manuel Blum and W. Sakoda. On the capability of finite automata in 2 and 3 dimensional space. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. IEEE, 1977.

[26] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K. Warmuth. Occam's razor. *Information Processing Letters*, 24:377–380, April 1987.

[27] J. U Bowie, R. Lüthy, and D. Eisenberg. A method to identify protein sequences that fold into a known three-dimensional structure. *Science*, 253:164–170, 1991.

[28] C. Branden and J. Tooze. *Introduction to Protein Structure*. Garland Publishing, Inc., New York and London, 1991.

[29] M. Brown, R. Hughey, A. Krogh, I. S. Mian, K. Sjölander, and D. Haussler. Using Dirichlet mixture priors to derive hidden Markov models for protein families. In *International Conference on Intelligent Systems and Molecular Biology*, pages 47–55, 1993.

[30] P. A. Bullough, F. M. Hughson, J. J. Skehel, and D. C. Wiley. Structure of influenza hemagglutinin at the pH of membrane fusion. *Nature*, 371:37–43, 1994.

[31] C. Carr and P. S. Kim. A spring-loaded mechanism for the conformational change of influenza hemagglutinin. *Cell*, 73:823–832, May 1993.

[32] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, 1990.

[33] Thomas Dean, Dana Angluin, Kenneth Basye, Sean Engelson, Leslie Kaelbling, Evangelos Kokkevis, and Oded Maron. Inferring finite automata with stochastic output functions and an application to map learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 208–214, 1992.

[34] Xiaotie Deng, Tiko Kameda, and Christos H. Papadimitriou. How to learn an unknown environment. In *Proceedings of the 32nd Symposium on Foundations of Computer Science*, pages 298–303. IEEE, 1991.

[35] Xiaotie Deng and Christos H. Papadimitriou. Exploring an unknown graph. In *Proceedings of the 31st Symposium on Foundations of Computer Science*, volume I, pages 355–361, 1990.

[36] I. Dodd and J. B. Egan. Improved detection of helix-turn-helix DNA-binding motifs in protein sequences. *Nucleic Acid Research*, 18:5019–5026, 1990.

[37] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, Inc., New York, 1973.

[38] Gregory Dudek, Michael Jenkin, Evangelos Milios, and David Wilkes. Using multiple markers in graph exploration. In *SPIE Vol. 1195 Mobile Robots IV*, pages 77–87, 1989.

[39] Jack Edmonds and Ellis L. Johnson. Matching, Euler tours and the Chinese Postman. *Mathematical Programming*, 5:88–124, 1973.

[40] D. Fass and P. S. Kim. The envelope protein of moloney murine leukemia virus contains a three-stranded coiled coil: Structural similarity between retrovirus and influenza membrane-fusion proteins. Submitted for publication, 1995.

[41] Paul Fischer and Hans Ulrich Simon. On learning ring-sum-expansions. *SIAM J. Computing*, 21(1):181–192, February 1992.

[42] V. Fischetti, G. Landau, J. Schmidt, and P. Sellers. Identifying periodic occurrences of a template with applications to protein structure. *Information Processing Letters*, 45(1):11–18, 1993.

[43] Yoav Freund, Michael Kearns, Dana Ron, Ronitt Rubinfeld, Robert Schapire, and Linda Sellie. Efficient learning of typical finite automata from random walks. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, pages 315–324, May 1993.

[44] Sally Goldman. *Learning Binary Relations, Total Orders, and Read-Once Formulas*. PhD thesis, MIT Dept. of Electrical Engineering and Computer Science, September 1990. (MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-483, July 1990).

[45] Irving John Good. *The Estimation of Probabilities*. The MIT Press, Cambridge, MA, 1965.

[46] M. Gribskov. Translational initiation factors IF-1 and eIF-2α share an RNA-binding motif with prokaryotic ribosomal protein S1 and polynucleotide phosphorylase. *Gene*, 119:107–111, 1992.

[47] L. Holley and M. Karplus. Protein secondary structure prediction with a neural network. *Proceedings of the National Academy of Sciences*, 86:152–156, 1989.

[48] Ming-Yang Kao, Yuan Ma, Michael Sipser, and Yiqun Yin. Optimal constructions of hybrid algorithms. In *Symposium on Discrete Algorithms*. SIAM, 1994.

[49] Ming-Yang Kao, John Reif, and Stephen Tate. Searching in an unknown environment: An optimal randomized algorithm for the cow-path problem. In *Symposium on Discrete Algorithms*. SIAM, 1993.

[50] Michael Kearns, Ming Li, Leonard Pitt, and Leslie Valiant. On the learnability of boolean formulae. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, pages 285–295, New York, New York, May 1987.

[51] Rolf Klein. Walking an unknown street with bounded detour. *Computational geometry: theory and applications*, 1(6):325–351, June 1992. Also published in The 32nd Symposium on Foundations of Computer Science, 1991.

[52] Jon Kleinberg. On-line algorithms for robot navigation and server problems. Master's thesis, MIT Department of Electrical Engineering and Computer Science, May 1994. (Published as MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-641).

[53] A. Krogh, M. Brown, S. Mian, K. Sjölander, and D. Haussler. Hidden Markov models in computational biology: Applications to protein modeling. Technical Report UCSC-CRL-93-32, University of California at Santa Cruz, 1993.

[54] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2:285–318, 1988.

[55] D. Liu, J. Bienkowska, C. Petosa, R. J. Collier, H. Fu, and R. Liddington. Crystal structure of the zeta isoform of the 14-3-3 protein. *Nature*, 376, July 13, 1995.

[56] M. Lu, S. Blacklow, and P. S. Kim. A trimeric structural domain of the HIV-1 transmembrane glycoprotein. *Nature Structural Biology*, 2(12), December 1995.

[57] Vladimir Lumelsky and Alexander Stepanov. Path-planning strategies for a point mobile automaton moving amidst unknown obstacles of arbitrary shape. *Algorithmica*, 2:403–430, 1987.

[58] A. Lupas, M. van Dyke, and J. Stock. Predicting coiled coils from protein sequences. *Science*, 252:1162–1164, 1991.

[59] R. Maclin and J. Shavlik. Refining algorithms with knowledge-based neural networks: improving the Chou-Fasman algorithm for protein folding. In Stephen Hanson, George Drastal, and Ronald Rivest, editors, *Computational Learning Theory and Natural Learning Systems*, pages 249–286. The MIT Press, 1994.

[60] S. Muggleton, R. D. King, and M. Sternberg. Protein secondary structure prediction using logic-based machine learning. *Protein Engineering*, 5(7):647–657, 1992.

[61] Christos H. Papadimitriou. On the complexity of edge traversing. *J. Assoc. Comp. Mach.*, 23:544–554, 1976.

[62] Christos H. Papadimitriou and Mihalis Yanakakis. Shortest paths without a map. *Theoretical Computer Science*, 84:127–150, 1991.

[63] D. A. D. Parry. Coiled coils in alpha-helix-containing proteins: analysis of residue types within the heptad repeat and the use of these data in the prediction of coiled-coils in other proteins. *Bioscience Reports*, 2:1017–1024, 1982.

[64] Leonard Pitt and Leslie G. Valiant. Computational limitations on learning from examples. Technical report, Harvard University Aiken Computation Laboratory, July 1986.

[65] Leonard Pitt and Leslie G. Valiant. Computational limitations on learning from examples. *Journal of the ACM*, 35(4):965–984, 1988.

[66] Leonard Pitt and Manfred K. Warmuth. Reductions among prediction problems: On the difficulty of predicting automata (extended abstract). In *3rd IEEE Conference on Structure in Commplexity Theory*, pages 60–69, Washington, DC, June 1988.

[67] N. Qian and T. Sejnowski. Predicting the secondary structure of globular proteins using neural network models. *Journal of Molecular Biology*, 202:865–884, 1988.

[68] Nagewara S. V. Rao, Srikumar Kareti, Weimin Shi, and S. Sitharama Iyengar. Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms. Technical Report ORNL/TM-12410, Oak Ridge National Laboratory, July 1993.

[69] Ronald Rivest. Machine learning lecture notes, 1994.

[70] Ronald L. Rivest and Robert E. Schapire. Inference of finite automata using homing sequences. *Information and Computation*, 103(2):299–347, April 1993.

[71] Dana Ron and Ronitt Rubinfeld. Exactly learning automata with small cover time. In *Proceedings of the 1995 Conference on Computational Learning Theory*, Santa Cruz, CA, July 1995.

[72] Robert H. Sloan. *Computational Learning Theory: New Models and Algorithms*. PhD thesis, MIT EECS Department, May 1989. (Published as MIT/LCS/TR-448.).

[73] R. Tatusov, S. Altschul, and E. Koonin. Detection of conserved segments in proteins: Iterative scanning of sequence databases with alignment blocks. *Proceedings of the National Academy of Science*, 91:12091–12095, December 1994.

[74] Leslie G. Valiant. A theory of the learnable. *Communications of the ACM*, 27(11):1134–1142, November 1984.

[75] B. Xiao, S. Smerdon, D. Jones, G. Dodson, Y. Soneji, A. Aitken, and S. Gamblin. Structure of a 14-3-3 protein and implications for coordination of multiple signalling pathways. *Nature*, 376, July 13, 1995.