2009

# Locality and Complexity in Path Search

Andrew Hunter
*Harvey Mudd College*

# Locality and Complexity in Path Search

**Andrew H. Hunter**

Nick Pippenger, Advisor

Ran Libeskind-Hadas, Reader

May, 2009

HARVEY MUDD COLLEGE

Department of Mathematics

# Abstract

The path-search problem considers a simple model of communication networks as channel graphs: directed acyclic graphs with a single source and sink. We consider each vertex to represent a switching point, and each edge a single communication line. Under a probabilistic model where each edge may independently be free (available for use) or blocked (already in use) with some constant probability, we seek to efficiently search the graph: examine (on average) as few edges as possible before determining if a path of free edges exists from source to sink. We consider the difficulty of searching various graphs under different search models, and examine the computational complexity of calculating the search cost of arbitrary graphs.

# Acknowledgments

I would like to thank Professor Pippenger for his patience and boundless knowledge; Professor Hadas for his enthusiasm, support, and encouragement; and my father, without whom I never would have done a single piece of academic writing.

# Contents

# List of Figures

# Chapter 1

# Introduction

Under conditions of heavy traffic, communication networks can run short of useful channels between two parties wishing to interact, and even when they can, finding such channels can take more effort than is desirable. Both of these problems stem from a given network's topology, which both determines if it will contain the desired paths, and how hard finding those paths actually is. Thus, we need to find better network topologies, that are highly likely to remained linked under traffic, and easily admit algorithms for finding communication channels.

To do this, we need good models for both of these criteria (the likelyhood that the paths we wish to find exist, and the cost incurred in finding those paths.) Path search is just such a model. In this work, we consider two previously unstudied facets of path search:

- A more realistic restriction of the standard model.
- Several complexity-theoretic results on the difficulty of analyzing networks under the path search model.

## 1.1 Path Search & Notation

### 1.1.1 Definitions

The *path-search* problem gives a directed acyclic graph $G$, with a single source $s$ and sink $t$, and a probability $q$. Independently let each $e \in E$ be either *free* (sometimes *available*) with probability $q$ or *blocked* (sometimes *busy*) with probability $p = 1 - q$; however, assume that at the start, we do not know the status of any of the edges. On average, how many edges must we *probe* (query the free or blocked status of) to either find a *free path* or a *busy cut*? We begin by defining some terminology and notation.

- A *channel graph* is a graph to which we can apply the path search problem—as above, this means it's a DAG with a single source and sink.
- The *vacancy probability*, denoted $q$, is the probability that any single edge in a channel graph is free.
- The *occupancy probability*, denoted $p = 1 - q$, is the probability that any single edge is blocked.
- A *free path* is a path consisting of only free edges. We say a channel graph is *free* or *linked* if it contains a free path.
- The *free probability* of $G$, denoted $P_f(G, q)$ is the probability that $G$ is free if $q$ is the vacancy probability.
- A *busy cut* is a cut separating $s$ and $t$, consisting of only busy edges. We say a channel graph is *blocked* if it contains a busy cut.

  *Remark* 1.1. Any channel graph is either free or blocked. Either there exists a free path from $s$ to $t$, or we can take at least one blocked edge from every path from $s$ to $t$, and construct a busy cut.
- The *blocking probability* of $G$, denoted $P_b(G, q)$ is the probability that $G$ is blocked.

  *Remark* 1.2. Of course, for all $q$ and $G$,

  $$P_f(G, q) + P_b(G, q) = 1,$$

  since any graph is either free or blocked.
- The *search cost* of a channel graph $G$ is the expected number of probes we must make before we can conclude that $G$ is free or blocked (by demonstrating a fully probed free path or busy cut.) We denote this value by $E(G, q)$, and sometimes $E_A(G, q)$ for the expected number of probes taken by some search algorithm $A$ (see Section 1.2). $E(G, q)$, then, is just the minimum of $E_A(G, q)$ over all algorithms $A$.

### 1.1.2 Motivation

A good interpretation of the intuition behind path search is to see channel graphs as communication networks—imagine we want to let $s$ talk to $t$, but have limited communication bandwidth. Each vertex in the channel graph represents a switch in our network, and each edge a communication channel, that may or may not be in use. In this context, the path search question becomes: how long does it take to determine if we can let $s$ talk to $t$?

This is obviously a very simple model, and most of the time, we ignore this explanation. We will usually consider the problem on abstract graphs, and not worry about the realistic interpretation. However, using this model can help provide understanding of what's going on; also, we will soon consider more complicated cases of path search (see 1.3) which we will justify in terms of the network analogy.

## 1.2 Search Algorithms

In our definitions, we described the search cost of channel graphs in terms of "search algorithms." These are our central objects of study—our main goal is to examine the search cost of various graphs—and thus we should carefully define these algorithms, so we can accurately characterize their costs.

### 1.2.1 Formalizing Path Search Algorithms

There are a number of ways we can consider path search algorithms. All of them can be expressed as a simple decision procedure based on marking and examining edges.

1. Begin by marking each edge of the graph as "unprobed".
2. Examine the graph:
   (a) If $G$ contains a path from $s$ to $t$ with each edge marked "free", halt and return "free".
   (b) If $G$ contains a cut that separates $s$ and $t$ with each edge marked "blocked", halt and return "blocked".
3. Use some function $f$ to select an edge in $G$. (Without loss of generality, we consider only those functions $f$ such that $e = f(G)$ is always

marked "unprobed". In other words, we only consider algorithms that never probe the same edge twice: doing so is always suboptimal.) Probe $e$; mark it as either "free" or "blocked", whichever's appropriate. (This step is the algorithm-dependent part, and finding the optimal algorithm boils down to choosing the best possible $f$.)

4. Go to Step 2.

Note that from a complexity-theoretic standpoint (but see Section 1.4), both of the termination tests are relatively easy (in P). Similarly, if we care, it's easy to actually give the path or cut that our algorithm found using standard graph-theoretic algorithms such as depth-first search (Cormen et al., 2001).

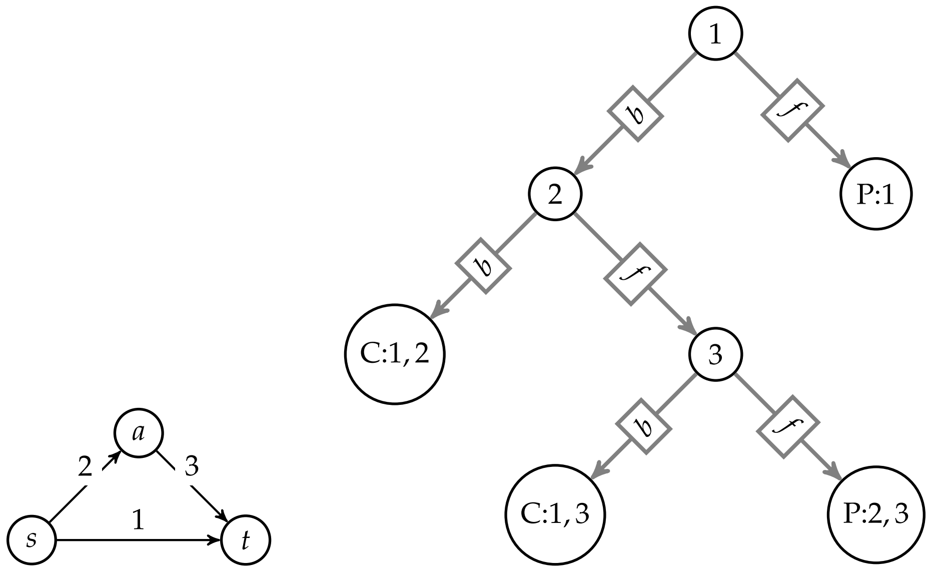
- For Step 2a, apply DFS from $s$ to the subgraph of $G$ with just those edges marked "free". If $t$ is reachable, we have found a path (and can recover it trivially from the DFS's forest.)
- For Step 2b, apply DFS from $s$ to the subgraph of $G$ with just those edges marked "free" or "unprobed". If $t$ is not reached, we know there's a busy cut. Consider the set $V_s$ of vertices reachable from $s$. The busy cut we found is exactly those edges from $V_s$ to $V - V_s$.

Given, then, that these tasks are mostly simple, we will generally (when evaluating an algorithm) not consider the computational cost of its decision making; only the average number of probes it makes. This choice is not without precedent; for example, see the discussion of certificate and decision-tree complexity in Arora and Barak (2009: ch. 12).

### 1.2.2 An Alternate Formalization

Since we are considering only deterministic algorithms,[1] if an algorithm receives the same feedback from a graph (i.e., is told the same results from its probes) on two runs, it will make the same decisions, probe the same edges, and come to the same conclusion in the same time. This determinism leads to another representation of our algorithms: decision trees. For

[1] An attentive reader will ask if ignoring randomness is safe—we're already in a randomized problem space, why not allow ourselves random choices? However, we are playing what is called a *game against nature* (see Section 1.4), and in such games it is well known that we can always meet the expectation of any *mixed* (random) strategy with a *pure* (deterministic) strategy (Papadimitriou, 1985). Thankfully, we do not need to consider the additional possibilities of randomized algorithms.



(a) A channel graph $G$.

(b) Decision tree for checking node 1 first, then going through nodes 2 and 3.

Figure 1.1: A graph $G$ and associated decision tree.

any channel graph $G$ and any algorithm (function $f$), we can write down a binary tree that represents that algorithm's decisions on that graph. Each internal node is labeled with an edge and has two children labeled "free" and "blocked". Each leaf is labeled with either a path or a cut. It's clear how the tree is built: each vertex represents a state and the action the algorithm will choose to take at that point. Under this model, the expected cost of the algorithm on the graph is just the average depth of the tree (with subtrees weighted by the blocking probability $p$). A simple graph and an associated decision tree for searching it are shown in Figure 1.1. The particular way we express our algorithms is unlikely to matter, but it is important have a way to formally consider the cost model, especially when we begin looking at complexity-theoretic problems in path search.

### 1.2.3 Graph Families and Search Complexity

In a certain sense, we are interested in the computational complexity of searching various graphs. However, of course, for any fixed graph $G = (V, E)$, there is a simple constant bound on the cost of searching $G$: we probe every edge, in some arbitrary order, then announce a decision. Since we speak entirely in average-case asymptotics, ours is an $O(1)$ algorithm: $|E|$ is a constant.

Obviously, we don't consider all graphs to have constant search complexity. Hence, we typically consider infinite families of graphs. We define a sequence $G_0, G_1, G_2, \ldots$ of graphs with some common kind of structure, but increasing size. (For the moment, our graphs will typically have paths that are $O(k)$ edges long, but $O(c^k)$ total edges.) We are then interested in some family of algorithms[2] that can search all the $G_k$. If we then define

$$E_G(k) = E(G_k, q)$$

we can then define the search complexity of the graph family $G_k$ as the asymptotic growth of $E_G$ with $k$. This complexity measure helps justify our choice of growth rates for path lengths and number of edges. Thus, an "efficient" (polynomial) algorithm for $G_k$ is meaningful. Unless the number of edges grows exponentially, every algorithm is efficient. Conversely, unless "short" paths exist, even an algorithm that always guessed the right edge to probe would be exponential just because it had to prove an entire path was free.

## 1.3 Local Path Search

Our main consideration here is a restriction of path search. Path search, as currently defined, does not mesh well with our network routing interpretation. If we only allow ourselves a subset of the original choices—specifically, we no longer allow fully global consideration of the graph—we may find that fewer graphs can be efficiently searched.

### 1.3.1 Is Global Searching Necessary?

We will see a number of search algorithms (for instance, see 2.2) that are efficient, but rely on being able to search wherever they need to. For example, consider a family $G_k$, shown in Figure 1.2. Here, $s$ has $2^k$ disjoint paths of length $k$, all leading to a single path of length $k$. For all $q < 1$,

$$\lim_{k\to\infty} P_B(G_k, q) = 1.$$

[2]Typically, we don't bother going into this much detail, but we are defining some sequence of functions $f_0 : G_0 \to e, f_1 : G_1 \to e, \ldots$ as we define $f$ in Section 1.2.1. Most often, we'll define some $f : \forall i. G_i \to e$, that describes the algorithm's choice on any of the graphs in the family.

More to the point, we can find this out with an (expected) constant number of probes in the "tail" of the graph (by using the path-greedy lemma, 1.3.) If we instead examine the "wide" front of the graph, we might examine a very large number of edges before finding a cut (or determining we can get through that part of the graph.[3]) Allowing algorithms to search wherever

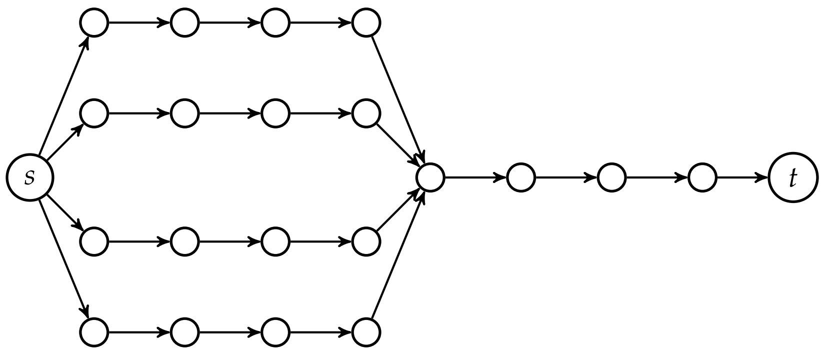


Figure 1.2: $G_2$, a graph where the location of search matters.

they please is perhaps an unrealistic assumption given our physical justification: if $s$ is trying to talk to $t$, how can $s$ start out by fiddling with the network near $t$—wouldn't it need to be able to talk to $t$ to do that?

So, our physical model suggests a logical restriction of the problem, to what we can call *local* path search: What if we can only probe edges we have access to? That is, at any step of the algorithm, we have probed some set of edges, and found some subset of them to be free. If we have found a free path from $s$ to some vertex $v$, then it is reasonable—in our physical model—to probe edges leaving $v$, as we clearly can communicate with $v$, and thus make decisions involving information from $v$'s vicinity. So, in local path search, we say that at any step, we can only probe an edge $(u, v)$ if there exists a free path from $s$ to at least one of $u$ or $v$ (but see Section 1.3.2).

Note that this restriction does not affect the decision power of our algorithms: if a free path from $s$ to $t$ exists, we will be able to probe it in order—each edge in turn will become available. Similarly, if there is no free path, we will eventually find a busy cut—with sufficient probes, we can find a path to every reachable vertex. If we determine (by there being no more available edges) that $t$ is not in that set, as discussed in 1.2, we have

[3]We won't bother finding the exact cost of searching $G_k$, locally or otherwise; we just illustrate with an example the idea that being able to search different parts of the graph can be very valuable.

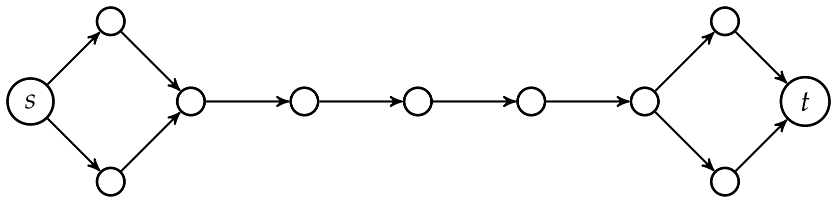


Figure 1.3: $GG_1$, where ends are not always enough.

found a busy cut. It may be considerably more expensive—in fact, cost is our first question—to search graphs locally, but we can always do it.

So, local path search is decidable, but is it efficient? We can show that for at least some graphs, it is asymptotically slower; see 2.3. In fact, this question is the main purpose of our work; our ultimate goal is to accurately characterize how the locality restriction slows search. We wish to find where we can search efficiently, where we cannot, and what the distinguishing characteristics of such graphs are.

### 1.3.2 Bridging the Local–General Gap

We will see that for many graphs, allowing general search—probing wherever the algorithm wants—is sufficient for the existence of efficient search algorithms, but allowing fully local search (probing from source-reachable vertices) prevents efficient search. However, this choice—all or nothing—is too harsh. We actually don't always need to be able to probe from anywhere. For example, for $G_k$ as above, all we need is the ability to probe from either end: probing on the narrow tail near $t$ will quickly find a small cut.

Is this true in general? No. For example, consider a graph family $GG_k$, defined as the composition of $G_k$ with the reversal of $G_k$, as shown in Figure 1.3. If we are only allowed $s$ and $t$, probing $GG_k$ will take exponential time. Otherwise, we could adapt the algorithm to locally probe $G_k$ (since we effectively are simultaneously locally probing two copies of $G_k$ from the front.) However, adding just one more vertex—the center—leaves us simultaneously probing two copies of $G_k$ from *both* ends, and we know doing so *is* efficient. This leads naturally to a simple question: given some graph $G$, what is the smallest set of vertices of $G$ we must be given "access" to search it efficiently?

We will write $E_S(G, q)$ for the optimal average cost of searching some graph $G$, with $S$ as the set of allowable starting vertices.

## 1.4 Complexity of Algorithm Generation

For the graph families and path search models we have considered so far, depth-first search of various forms has been an optimal algorithm. But we have no guarantee that DFS is always best—some graph $G$ might require a totally different search strategy.

One thing we can guarantee, however, is that we can determine (with some computational effort) what the optimal algorithm for any graph is. The obvious algorithm for doing so is

1. If $G$ contains a marked free path or busy cut, the cost of searching $G$ is zero and we don't need to probe anything.
2. Otherwise, for all $e \in E$,
    (a) If $e$ is already marked, or $e$ isn't accessible from our currently available vertices,[4] continue to the next $e$.
    (b) Otherwise, mark $e$ free, recursively use this algorithm to compute the cost of probing that graph, and call the result $e_f$.
    (c) Then, mark $e$ blocked, recursively use this algorithm to compute the cost of probing that graph, and call the result $e_b$.
    (d) The optimal cost of searching $G$ by starting with $e$ is then $c_e = qe_f + pe_b$.
3. The optimal edge to take is the one with the lowest $c_e$; the cost is $c_e + 1$.

This algorithm works, but is slow. It considers the cost of probing edges in every possible order, for a cost of $O(n!)$ time.[5] We note that the recursion depth of this algorithm is at most polynomial, as there are at most $n^2$ edges in a graph of size $n$, and we only recurse after assigning a mark to an edge. So, this algorithm only takes polynomial space. Can we do better? In complexity-theoretic terms, we want to know the complexity of deciding this language:

$$PATHCOST = \{< G, S, x, q > | E_S(G, q) \leq x\}.$$

[4]Note that step this means we can make this algorithm work for general path search, local search, or search from some set of allowed vertices.

[5]in fact, it does more than this, but that aspect is enough to show that it takes intractable time.

The above algorithm proves that $PATHCOST \in PSPACE$. We would like to show that either $PATHCOST$ is $PSPACE$-complete, or that we can decide the language with fewer resources—ideally that $PATHCOST \in P$, or failing that, perhaps $PATHCOST \in PH$ (still an improvement) or the like.

We can see evidence that $PATHCOST$ might be $PSPACE$-complete in the structure of path search: it is a *game against nature*, a two-player game with a "disinterested" opponent who plays entirely randomly. $PSPACE$ can be seen as the class of problems expressible as the optimal strategy for a two-player game; consider the canonical $PSPACE$-complete problem $TQBF$, (Arora and Barak, 2009). It asks if a quantified boolean expression,

$$\exists x_1 \forall x_2 \exists x_3 \forall x_4 \ldots \exists x_{p(n)-1} \forall x_{p(n)} \phi(x_1, \ldots, x_{p(n)}),$$

where $\phi$ is a boolean expression with a polynomial $p(n)$ number of variables, is satisfiable. We can see proving or disproving this formula as a game between two players: "forall" and "exists". They take turns selecting values for variables, and "exists" wins at the end if the expression is true. Hence, it is unsurprising that finding optimal strategies for several real-world games, such as Othello, are $PSPACE$-complete (Iwata and Kasai, 1994).

In fact, we can extend $PSPACE$-completeness to suitably complex games against nature; the computational power doesn't just come from both players playing optimally. Papadimitriou (1985) showed that the general class of polynomial-length games against nature is equal to $PSPACE$, and demonstrated a number of specific $PSPACE$-complete games against nature. One worth considering is *dynamic graph reliability*.

In the dynamic graph reliability problem, we are given a directed acyclic graph with a source $s$ and a sink $t$, and wish to travel from $s$ to $t$. However, if we are currently at a vertex $v$, and wish to travel along an edge $e$, there is a probability $p(e, v)$ that that edge will fail (become unavailable for use) before our next move. We wish to find the maximal probability of successful traversal.

Note that path search can be encoded as a form of dynamic graph reliability problem! It is not a perfect match, and the reverse isn't true (it is not immediately obvious how to encode any dynamic graph reliability problem as a $PATHCOST$ instance) but it is easy to see the similarity: each move we try to make corresponds to an edge probe with some probability of failure, and both problems seek to find a working path from $s$ to $t$.

Thus one plausible goal is to find a reduction from dynamic graph reliability to $PATHCOST$ or a related path search problem. Of course, $PATHCOST$ is in some ways simpler (or just different) from graph reliability: it is not out of the question that $PATHCOST$ is not $PSPACE$-hard and we can find a polynomial time algorithm for it,[6] or some other improvement (such as a containment in some level of the polynomial hierarchy).

## 1.5 Techniques and Results

This section is primarily dedicated to proving useful lemmas and results for analyzing search algorithms, and describing the basis of the techniques we will use heavily.

### 1.5.1 Forcing Information

A useful technique for analyzing lower bounds on searching graphs is what we will call *forcing information* on an algorithm. Since we are concerned with average case asymptotics, we can restrict our attention to the cost of searching the graph in some common case. That is, suppose that for some graph family $G_k$, some probalistic event $X$ (for example, the graph is free) occurs with positive probability for all $k$. Then (if we like) we can derive a lower bound just from the expected cost of searching the graph given that $X$ happens.

In fact, we can take our analysis to assume that the cost of searching the graph is zero if $X$ doesn't happen; since X occurs with positive probability—that is, we have some $\epsilon$ such that for all $k$,

$$P(X) \geq \epsilon > 0$$

So if the expected cost of searching the graph when $X$ occurs is $f(k)$, the overall cost is bounded as

$$E(G_k, q) > (1-\epsilon) \cdot 0 + \epsilon \cdot f(k) = \epsilon f(k) \in \Omega(f(k)).$$

This fact will be helpful when the minimum cost for any algorithm is easily seen (and high) when $X$ occurs, but not in general.

We can see the effect of this technique by looking at the decision tree representation of any algorithm. We simply prune out any subtree where a probe resulted in a contradiction of $X$ (e.g., if $X$ is "the graph is free", we

[6]Of course, $PATHCOST$ being $PSPACE$-hard doesn't rule that out...but since that would prove $P = PSPACE$, I'm not exactly holding my breath.

prune out any edge leading to a declaration of a cut). Our interpretation of this change is that when an algorithm makes a probe that might contradict $X$, we ensure (with high probability) that it gets the answer that preserves the possibility of $X$. This pruning will shrink the decision tree, but if we take the same weighted average, we can still discover useful lower bounds. In fact, we will often be able to ignore the cost of "proving" $X$; the number of edges probed after any algorithm determines $X$ will be enough to get the bound we want.

### 1.5.2 Path-Greedy Algorithms

When determining lower bounds on locally searching graphs, it will be helpful to restrict our attention to algorithms that in some sense "focus" on some small part of the graph at any one point in time. One such useful class are *path-greedy* algorithms, which we define as algorithms that can be described as

1. Pick a path from some available vertex $v$ to $t$ through $G$.
2. Probe edges along that path, starting at $v$, continuing until some edge is blocked (and the path is no longer of value).
3. Repeat until the graph is decided.

In other words, once these algorithms choose a path to examine *any* part of (the first edge out of the available vertex), they focus entirely on that path until it is of no use. The path-greedy lemma, which tells us that we only need consider such algorithms (if we want), will be useful.

**Lemma 1.3.** *For any channel graph $G$, there exists an asymptotically optimal path-greedy search algorithm.*

*Proof.* We can simulate an arbitrary local search algorithm with a path-greedy algorithm, with a constant $(1/p)$ factor overhead. Pick some optimal algorithm for locally searching $G$. At each step, where the original algorithm would probe an edge $e$, pick a remaining path involving $e$ (obviously, one exists, or there would be no point in probing $e$). Probe that path instead of just $x$. Clearly we have enough information to simulate the original algorithm—we can just ignore the knowledge we have about the additional edges we probed, if necessary. Moreover, this method is cheap: we probe the first edge on the path, which is $e$ (since the algorithm is local, the source of $e$ is accessible, and thus the path including it starts there).

Then, if that edge is free (with probability $q$) we probe the next, and so on. If the path has $n$ edges in total, the number of edges we expect to probe is:

$$1 + q\,(1 + q\,(1 + \dots)) = 1 + q + \dots + q^{n-1} < 1 + q + q^2 + \dots = \frac{1}{p}$$

and so we can efficiently use a path-greedy algorithm. $\square$

In many cases, path-greedy algorithms are obviously expensive; the point of this lemma is that so long as we can prove that all path-greedy algorithms satisfy some lower bound, we do not have to worry that some algorithm outside of our restricted class will do (substantially) better.

# Chapter 2

# Analyzing Fully Parallel Graphs

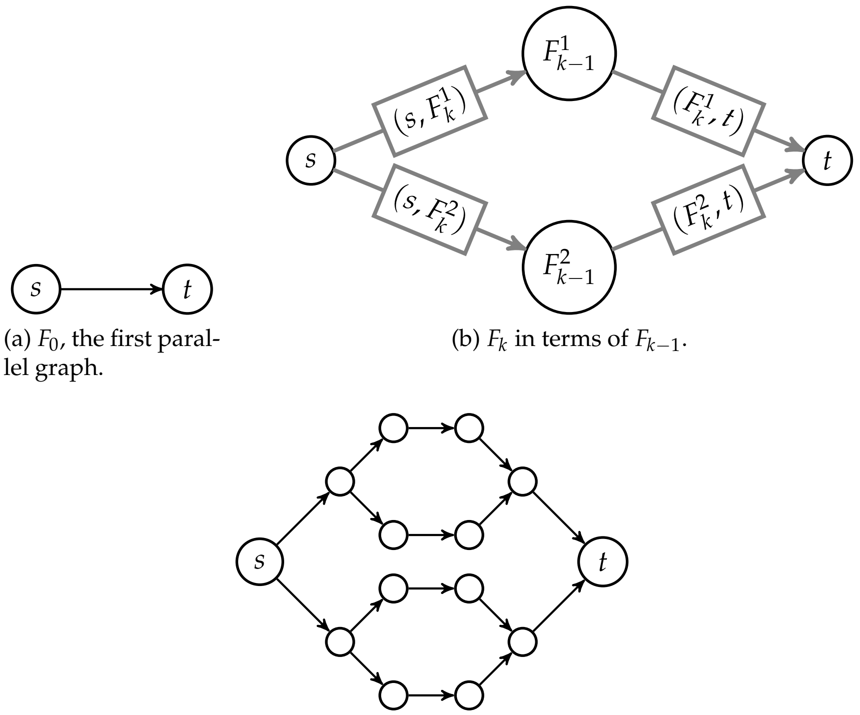
A relatively simple family of graphs, called the *fully parallel graphs*, will be our first subject of study; in particular, they can teach us a great deal about the differences in cost between global and local path search, while having a tractable analysis.

## 2.1 The Fully Parallel Graphs

The fully parallel graphs, denoted $F_k$, have a simple recursive definition shown in Figure 2.1. $F_0$ is a single edge connecting $s$ and $t$, and $F_k$ is two copies of $F_{k-1}$, placed in parallel with one another.

For convenience, we will need to label the two contained subgraphs. As shown in Figure 2.1, we label one of the included copies of $F_{k-1}$ as $F_k^1$; the other as $F_k^2$. We write $(s, F_k^1)$ for the single edge from $s$ into $F_k^1$, and the like for $t$ and $F_k^2$.

We can also define $F_k$ differently, as the composition of two complete binary trees, as seen in Figure 2.2. Here, $T_k$ is a complete binary tree of depth $k$. Then, $F_k$ is two copies of $T_k$, with the leaves of one linked to the other (importantly, they are linked "in order", as shown, respecting the symmetry of the graph). We will call the tree rooted at $s$ the *diverging* or *initial* tree, and the tree rooted at $t$ the *converging* or *final* tree. This notation will occasionally be more convenient when analyzing certain algorithms.



(a) $F_0$, the first parallel graph.

(b) $F_k$ in terms of $F_{k-1}$.

(c) $F_2$, an example of a larger parallel graph.

Figure 2.1: The definition of fully parallel graphs.

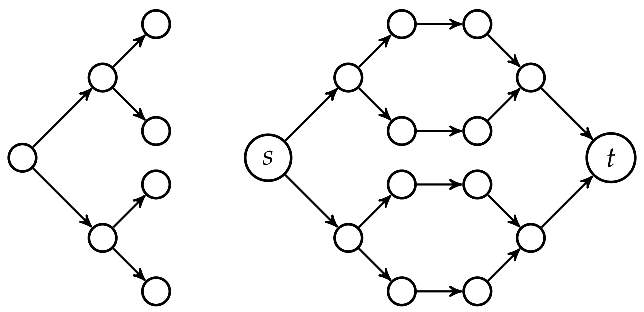
## 2.2 Bounds on Parallel Graphs

From previous work (Pippenger, 1999), we know that if $0 \leq q < \frac{1}{\sqrt{2}}$ then,

$$\lim_{k \to \infty} P_b(F_k, q) = 1.$$

We can see this fact via the union bound: $F_k$ is the union of $2^k$ paths of length $2k+1$, and all edges on a path must be free (with probability $q$) for the graph to be free. Thus,

$$P_f(F_k, q) \leq 2^k q^{2k+1} = q(2q^2)^k.$$



(a) $T_2$, a complete binary tree of depth 2.

(b) $F_2$, two connected copies of $T_2$.

Figure 2.2: An alternate definition of fully parallel graphs.

Thus if $q < \frac{1}{\sqrt{2}}$, this bound goes to zero with increasing $k$. But if $\frac{1}{\sqrt{2}} \leq q \leq 1$, we have:

$$\lim_{k\to\infty} P_b(F_k, q) = \frac{(1-q^2)^2}{q^4}.$$

Thus, if $q$ is greater than the *critical probability* (here, $\frac{1}{\sqrt{2}}$), $P_b$ is bounded strictly away from 1 regardless of $k$; even in large fully parallel graphs, if $q$ is high enough, it is likely that the graph is free. This bound comes from "folding" $F_k$ along its midline into a binary tree.

We say that an edge in this tree exists if both edges it corresponds to in $F_k$ are free, with probability $q^2$. The graph is free if a leaf of depth $k$ exists, and the bound can be derived from treating the tree's edges as a branching process.

Moreover, we can search fully parallel graphs relatively efficiently *if* we do so globally.

**Theorem 2.1** (Lin and Pippenger (1996))**.** *For all* $0 \leq q \leq 1$,

$$E(F_k, q) \in O(k).$$

*Proof.* We demonstrate a simple algorithm: DFS from both ends simultaneously. (Note that this algorithm corresponds to searching the folded tree described above.)

1. Probe $(s, F_k^1)$. If that edge is free, probe $(F_k^1, t)$. If that edge is free, use this algorithm recursively to probe $F_k^1$. If that graph is free, $F_k$ is free.

2. If Step 1 didn't find a path, do the same with $F_k^2$; if there is no path there either, the graph is blocked.

We label the expected cost on $F_k$ of this algorithm as $E_{DFS}(k)$.

The expected cost $c(k)$ of the two steps 1 and 2 above are precisely the same (as they perform the same steps on identical subgraphs:)

$$c(k) = 1 + q(1 + qE_{DFS}(k-1)).$$

Furthermore, we will always execute 1, but we will only go to 2 if that didn't find a path, with probability

$$p(k) = 1 - q^2 P_f(F_k, q).$$

Then we can write the expected cost of this algorithm as

$$E_{DFS}(k) = (1 + p(k))c(k). \tag{2.1}$$

Let $d = (1 + p(k))(1 + q)$ (for our purposes, we really just want a constant with regards to $k$ upper bound on $d$; since $p(k) \leq 1$, we can use $d = 2 + 2q$). Then we can rearrange 2.1 to

$$E_{DFS}(k) \leq d + (1 + p(k))q^2 E_{DFS}(k-1).$$

Let $X(q) = \lim_{k\to\infty}\left((1 + p(k))q^2\right)$. Then we have

$$E_{DFS}(k) \leq d + X(q)E_{DFS}(k-1). \tag{2.2}$$

Interpret Equation2.2 as a geometric series with ratio $X(q)$. So, if $X(q) < 1$, the expected cost is bounded by a constant; if $X(q) \leq 1$, the cost is bounded by $O(k)$. Two cases are worth considering:

1. $q < \frac{1}{\sqrt{2}}$.

   Remember that $p(k) \leq 1$ regardless of $k$ or $q$ (since it is a probability). So
   $$X(q) \leq 2q^2 < 1.$$
   Thus in this range, the expected cost to search $F_k$ is *constant in k!*

2. $\frac{1}{\sqrt{2}} \leq q < 1$

   In this range, remember that
   $$\lim_{k\to\infty} P_f(F_k, q) = 1 - \frac{(1-q^2)^2}{q^4}.$$

Hence we have

$$\lim_{k\to\infty} p(k) = 1 - q^2 + \frac{(1-q^2)^2}{q^2} = 1 - q^2 + \frac{1}{q^2} - 2 + q^2 = \frac{1}{q^2} - 1,$$

which tells us

$$X(q) = (1 + \frac{1}{q^2} - 1)q^2 = 1.$$

So in this range, the expected cost to search $F_k$ is proportional to $k$.

Hence, for all $q$, $X(q) \leq 1$ and $E(F_k, q) \in O(k)$. □

While it is not obvious that this is a strictly optimal algorithm, this $O(k)$ bound is "good enough": each path in $F_k$ has length $2k+1$ (remember, also, that $F_k$ has $O(2^k)$ vertices—these paths are *short* compared to the size of the graph) and thus we're determining if a path exists (and finding one if so!) in about as much time as it takes to walk down one path. From a complexity-theoretic viewpoint, we simply can't do enough better for it to be worth trying.

In addition, Lin and Pippenger (1996) have demonstrated that this algorithm *is* asymptotically optimal, and their method is worth considering.

**Lemma 2.2** ((Lin and Pippenger, 1996))**.** *The above algorithm uses at most $\frac{1}{p}$ times as many probes as any algorithm for searching $F_k$.*

*Proof.* (Sketch.) We can demonstrate that the above algorithm is optimal among algorithms that work symmetrically from both ends (at each step, probing both ends of some subgraph we have demonstrated can be accessed from both source and target). Furthermore, *any* algorithm can be simulated with minimal overhead by such an algorithm. If the arbitrary algorithm would probe an edge $x$ at some step, instead probe both ends of the subgraph containing $x$ repeatedly until we either reach $x$ or have found a blocked edge that invalidates $x$. It is not hard to see that this process takes at most

$$1 + q + q^2 + \cdots < \frac{1}{p}$$

probes, and gives at least as much information as probing $x$. Thus, a suitably symmetric algorithm can do at least a factor of $\frac{1}{p}$ as well, and DFS does better than any symmetric algorithm. (This approach is a global-search variant of the path-greedy lemma, applicable to any graph where *any* edge is on exactly one path from $s$ or to $t$ (though not necessarily both.) □

## 2.3 Local Search on $F_k$

While the fully parallel graphs are efficiently globally searched without too much difficulty, the same is not true in the case of local search.

**Theorem 2.3.** *For local path search on the fully parallel graphs, we have (for $q > \frac{1}{2}$)*

$$\begin{aligned} E(F_k, q) &\in \Omega(c^k) \\ E(F_k, q) &\in O(C^k) \end{aligned}$$

*for some* $1 < c \leq C < 2$. *We know that*

$$(2q)^{-\log q} \leq c \leq C \leq \begin{cases} 2q & \text{if } q < \frac{1}{\sqrt{2}} \\ \frac{1}{q} & \text{otherwise} \end{cases}.$$

*Proof.* First we will demonstrate that we can perform the search with exponential cost. Then we demonstrate that any algorithm must satisfy some exponential lower bound.

### Achieving the Bound

We begin by demonstrating an algorithm that achieves this cost: depth-first search from *one* end. To reiterate

1. To search $F_0$, just probe the single edge there.
2. To search $F_k$, first probe $(s, F_k^1)$ (the choice here is obviously arbitrary). If the edge is free, recursively apply DFS to $F_k^1$; if that graph is free, probe $(F_k^1, t)$; if that edge is free, the whole graph is free and we're done.
3. Otherwise, repeat exactly the same procedure with $F_k^2$; if that method fails to find a path to $t$, the graph is blocked.

We can analyze the cost of this algorithm much as we did for the two-sided DFS in the general case. Again, we have to look at either one side or two sides (with the same expected cost per side):

$$E_{DFS}(k) = (1 + p(k))c(k),$$

where $c(k)$ is the cost of probing one side of the graph, and $p(k)$ is the probability we'll find that first side is blocked and we'll have to look at another. In fact, $p(k)$ has the same recursive definition. But here, $c(k)$ is different:

$$c(k) = 1 + q\left(E_{DFS}(k-1) + P_f(F_{k-1}, q)\right)$$

This cost is higher! The issue is that since we don't look at both $(s, F_k^1)$ and $(F_k^2, t)$ before diving recursively, we're considerably more likely to recur. With a similar rearrangement as before, we can write, for two constants $d_1$ and $d_2$,

$$d_1 + (1 + p(k))qE_{DFS}(k-1) \leq E_{DFS}(k) \leq d_2 + (1 + p(k))qE_{DFS}(k-1).$$

Hence defining as before,

$$X(q) = \lim_{k\to\infty}\left((1 + p(k))q\right).$$

We see we are bounded on both sides by a geometric series with ratio $X(q)$. We consider four cases:

1. $q < \frac{1}{2}$.

   Here we note that $p(k) \leq 1$, and see that $X(q) \leq 2q < 1$. So, in this range, we still see a constant expected cost.

2. $q = \frac{1}{2}$.

   Here we are still in the blocking regime and see $p(k)$ go to 1 at high $k$. Then we see $X(q) = 2q = 1$, and we find an expected cost of $O(k)$. Note this result is strictly worse than the general case, though still "tractable".

3. $\frac{1}{2} < q < \frac{1}{\sqrt{2}}$

   Here we still see $p(k)$ go to 1, giving us $X(q) = 2q > 1$, and the exponential growth we wanted.

4. $\frac{1}{\sqrt{2}} \leq q$.

   Here, $p(k)$ (as before) goes to $\frac{1}{q^2} - 1$, giving us

   $$X(q) = (1 + p(k))q = \frac{1}{q},$$

   as desired.

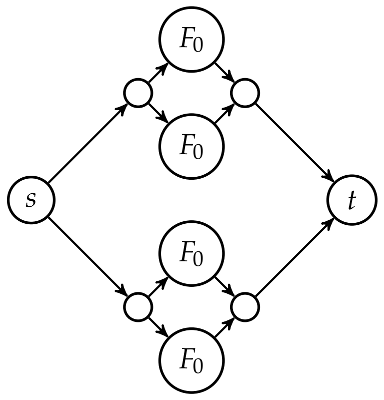


Figure 2.3: Breaking up $F_2$ into $T_2$, 4 copies of $F_0$, and $T_2$. If the internal subgraphs grow quickly enough, as $k$ does, the probability of any of them being free will go to zero.

**An Exponential Lower Bound Exists**

Of course, demonstrating that DFS takes exponential time doesn't show that nothing else is better—but we can demonstrate that an exponential lower bound applies to any algorithm.

**Lemma 2.4.** *For all $\frac{1}{2} < q < \frac{1}{\sqrt{2}}$ (the blocking range) there exists $c > 1$ such that we have*

$$E(F_k, q) \in \Omega(c^k).$$

*Specifically,*

$$E(F_k, q) \in \Omega\left(\left(2^{\frac{1}{2}\left(3+\frac{\log 2}{\log q}\right)} q\right)^k\right). \tag{2.3}$$

*Proof.* We demonstrate this bound by breaking the graph into two regions: the *tree* and the *barrier*. Specifically, we demonstrate that for some $d$ dependent only on $q$, breaking $F_k$ into a tree of depth $k/d$ rooted at $s$, another tree rooted at $t$, and the $2^{k/d}$ copies of $F_{k-k/d}$ between those trees allows us to bound the search with high probability to within the first tree—and furthermore, that searching that tree is expensive. Figure 2.3 demonstrates breaking up $F_k$; the key idea is that we can force every one of the central subgraphs being blocked on the algorithm.

**Lemma 2.5.** *Let $d > \frac{2\log q}{\log 2+2\log q}$. Then the probability as $k$ goes to infinity that $2^{k/d}$ copies of $F_{k-k/d}$ are all blocked goes to 1.*

*Proof.* Let $c = k/d$. If $P$ is the probability all copies of the subgraph are blocked, we have

$$P = \left(1 - P_f(F_{k-c})\right)^{2^c}.$$

Applying the union bound approximation for $P_f$ and the binomial theorem,

$$\begin{aligned} P &\geq 1 - 2^c q(2q^2)^{k-c} \\ &= 1 - q2^k q^{2k-2c} \\ &= 1 - q2^k q^{2k(d-1)/d} \\ &= 1 - q(2q^{2(d-1)/d})^k. \end{aligned}$$

So long as $r = 2q^{2(d-1)/d} < 1$, this probability goes to 1. But using the definition of $d$ (taking the bound as equality)

$$r = 2q^{2\frac{d-1}{d}},$$

so

$$\begin{aligned} \log r &= \log 2 + 2\frac{d-1}{d}\log q \\ &= \log 2 - 2\frac{\log 2}{2\log q}\log q \\ &= \log 2 - \log 2 = 0. \end{aligned}$$

Thus

$$r = 1.$$

□

Since the probability of the barrier being blocking goes to 1 with high $k$, it's bounded away from zero for all $k$ by a constant; we can force the algorithm to find all of the subgraphs in the barrier to be blocked.

Then, we must search all of the diverging tree to find a cut—we must get to each (available) leaf of that tree, so we can search the corresponding barrier subgraph and determine it is impassable. Thus, we bound $E(F_k, q)$ from below by the cost of searching the initial tree.

**Lemma 2.6.** *Let $T_k$ be a tree of depth $k$. Looking at the cost of searching the tree* fully *(to find all available leaves), we have*

$$E(T_k, q) \in \Theta((2q)^k).$$

*Proof.* We have $E(T_0, q) = 0$. Searching any large tree is simple: we must probe both children of the root; if either child is free, we must probe the corresponding subtree (since we want to find all available leaves). Thus,

$$E(T_k, q) = 2 \cdot (1 + qE(T_{k-1}, q)) = 2 + (2q)E(T_{k-1}, q).$$

The desired bound immediately follows from treating this formula as a geometric series. □

We can now put the pieces together. We have

$$d = \frac{2\log q}{\log 2 + 2\log q}.$$

We can bound the cost of searching $F_k$ as:

$$E(F_k, q) \geq E(T_{k/d}, q) \in \Theta\left((2q)^{k/d}\right).$$

Thus we have a minimal ratio of

$$c = (2q)^{1/d}$$

and tedious arithmetic gives Equation 2.3. □

This bound is not tight—DFS has a significantly worse ratio—but it is exponential. We can derive a similar exponential bound in the passing regime.

**Lemma 2.7.** *For all $\frac{1}{2} < q < \frac{1}{\sqrt{2}}$ (the blocking range) there exists $c > 1$ such that we have*

$$E(F_k, q) \in \Omega(c^k).$$

*Specifically, there exists some $d > 1$ depending only on $q$ such that*

$$E(F_k, q) \in \Omega\left(\left(\left(\frac{1}{q}\right)^{1/d}\right)^k\right).$$

*Proof.* The proof is complex and takes three steps:

1. We demonstrate that with high probability, an exponential number ($(2q)^k$) of the leaves of the converging tree are reachable.

2. We show that if a tree of depth $k$ has some large number of reachable leaves (exponential in $k$), then it must contain a subdivision of a complete binary tree of depth comparable to $k$.

3. We prove a lower bound of $\Omega\left(\left(\frac{1}{q}\right)^k\right)$ on searching a complete tree of depth $k$ from the leaves.

**Lemma 2.8.** *On average, $q(2q)^k$ of the leaves of the converging tree of $F_k$ are reachable.*

*Proof.* Let $R(k)$ be the expected number of reachable leaves of $T_k$ (when searched from the root.) Then,

$$\begin{aligned} R(0) &= 1 \\ R(k) &= 2qR(k-1), \end{aligned}$$

where $(2q(1-q)+2q^2) = 2q$ is the expected number of reachable copies of $T_{k-1}$ from the root of $T_k$. We immediately derive

$$R(k) = (2q)^k.$$

So, $R(k)$ leaves of the diverging tree are reachable; since each reachable leaf must pass through one unprobed edge (with probability $q$) to become a reachable leaf of the converging tree, the expected number of successful leaves is:

$$qR(k) = q(2q)^k.$$

□

In effect, we have proved that a significant fraction of the converging tree is of interest. Next, we demonstrate that this result implies that a substantial complete tree must be navigated.

We say the *pebble number* $P(T)$ of any tree $T$ is the depth of the largest binary tree contained in $T$ as a minor. We have

$$P(T_0) = 0.$$

If $T$ is a tree whose root has one child, $T'$, then

$$P(T) = P(T')$$

and if $T$'s root has two children, $T_a$ and $T_b$, then

$$P(T) = \begin{cases} k+1 & P(T_a) = P(T_b) = k \\ \max(P(T_a), P(T_b)) & \text{otherwise.} \end{cases}$$

Another interpretation is that the pebble number is the number of registers needed to evaluate an expression with a given abstract syntax tree. For proofs of the above and substantial analysis of upper bounds on the pebble number, see Flajolet et al. (1979). We will demonstrate that the reachable section of the converging tree in the average case has a high pebble number, since it has many leaves.

**Lemma 2.9.** *Define $L_{p,k}$ as the maximum possible number of leaves of a tree with depth $k$ and pebble number $p$. We have*

$$L_{p,k} = \sum_{i=0}^{p} \binom{k}{i}.$$

*From there, we have for any constant $d > 2$,*

$$L_{k/d,k} < \left((de)^{\frac{1}{d}}\right)^k.$$

**Corollary 2.10.** *If the converging tree has at least $q(2q)^k$ leaves, its pebble number (for sufficiently large $k$) is at least:*

$$p = \frac{k}{d}$$

*For some constant $d > 1$, depending only on $q$.*

*Proof.* Note that we have

$$\begin{aligned} L_{k,k} &= 2^k \\ L_{0,k} &= 1 \\ L_{p,k} &= L_{p,k-1} + L_{p-1,k-1}. \end{aligned}$$

The first two cases are trivial:

1. The only depth-$k$ tree with pebble number $k$ is a complete tree.
2. The only tree with a pebble number of 1 is a single path with exactly 1 leaf.

The recurrence relation can be derived from the definition of the pebble number; a tree with depth $k$ and pebble number $p$ has as the children of its root:

- A single tree with pebble number $p$ and depth $k - 1$;

- Such a tree, plus another child with pebble number less than $p$ and depth $k-1$; or,
- Two trees with pebble number $p-1$ exactly.

However, the first case trivially has fewer leaves than the second; the third case is subsumed as well, since $L_{p,k}$ obviously increases with $p$.[1] Thus we have:

$$L_{p-1,k-1} + L_{p-1,k-1} \leq L_{p,k-1} + L_{p-1,k-1}.$$

So only the second case remains in the recurrence as given. The closed form solution

$$L_{p,k} = \sum_{i=0}^{p} \binom{k}{i}.$$

is immediate by induction.[2]

For the corollaries, note that if $2p < k$,

$$L_{p,k} = \sum_{i=0}^{p} \binom{k}{i} < (p+1)\binom{k}{p}.$$

Let $p = k/d$, for some $d > 2$. We now have

$$L_{p,k} < \left(\frac{k}{d}+1\right)\binom{k}{\frac{k}{d}} \leq \left(\frac{k}{d}+1\right)\left((de)^{1/d}\right)^{k}.$$

Note that

$$\lim_{d\to\infty} (de)^{1/d} = 1,$$

and that so long as

$$2q > (de)^{1/d},$$

we have that $R(k)$ grows faster than $L_{\frac{k}{d},k}$; thus, for large $k$ and suitable $d$,

$$R(k) > L_{\frac{k}{d},k}.$$

Our conclusion is that there exists some $d$, depending only on $q$, such that for large enough $k$, the converging tree of $F_k$ has pebble number at least $\frac{k}{d}$ with high probability. □

[1]We can just add leaves from the bottom of the tree until we increase the pebble number.

[2]The structure of our closed form for $L_{p,k}$ suggests that a good combinatorial proof is possible, though we have yet to find one.

So we now know that we can force a large complete binary tree into the reachable part of $F_k$. In fact, we can do something stronger: we can force a large binary tree, *rooted at the root of $F_k$'s converging tree*, on the algorithm. We do so by forcing the four edges $(s, F_k^1), (s, F_k^2), (F_k^1, t), (F_k^2, t)$ to be free (with probability $q^4 > 0$); we then apply the above logic to the two reachable copies of $F_{k-1}$ to find large binary trees in each half; we then have a large binary tree rooted at the base of the entire channel graph, which is necessary.

The tree we have found is only *contained* in the reachable part of the graph $G$—to be exact, the reachable part of the tree is a subdivision of the binary tree in question. However, we will apply lower bounds to searching complete trees to this graph. This is safe.

**Lemma 2.11.** *Let $T$ be a complete binary tree and $U$ be a tree that contains $T$ as a rooted minor. Then*

$$E(U,q) \geq E(T,q).$$

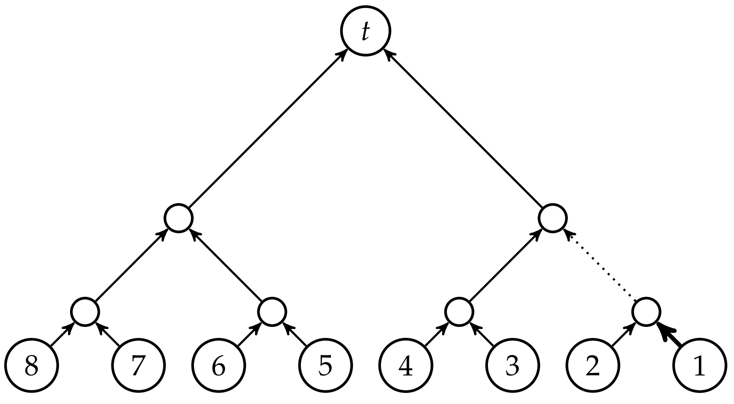
*Proof.* Any algorithm which searches $U$ also searches $T$. The only difference between $U$ and $T$ is that $U$ replaces single edges with possibly longer paths.

Since we will (see Lemma 2.12) prove a lower bound on searching binary trees by minimizing the number of leaves any step of a search can eliminate as plausible free paths, and moreover that the number of leaves eliminated by such a step shrinks if the search dies faster, the change to $U$ can only hurt. Since paths of length greater than one are *less* likely to be free, the search is only more likely to die at any one stage: the step in a search will eliminate fewer leaves, we must take more steps, and the cost is no lower. □

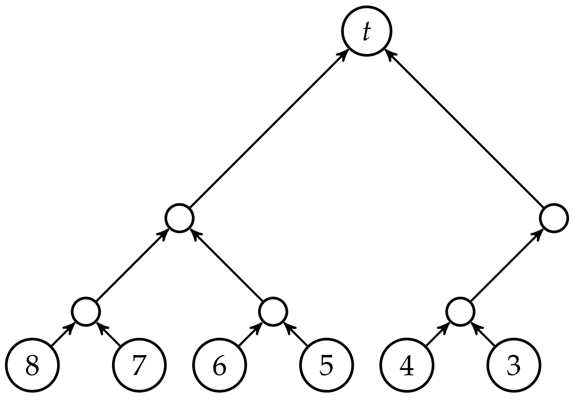
All that remains is to show that searching complete trees is expensive. This is relatively straightforward, but interesting.

**Lemma 2.12.** *Locally searching a complete binary tree $T_k$ from the leaves takes at least $\Omega\left(\left(\frac{1}{q}\right)^k\right)$ probes on average.*

*Proof.* We apply the path-greedy lemma, and thus only need consider algorithms that probe entire paths. Note that any path we can pick initially is of length $k$; moreover, after probing that path, we will still be left with only length $k$ paths! To see this, examine Figure 2.4. As soon as a path becomes blocked, all the shorter paths we have exposed by looking at it are pruned out, and all that remains are the (full length) paths disjoint from the part of the path we probed. So, we can consider searching a tree as a series of path-



(a) Searching one path in a tree of depth $k = 3$



(b) After that search, paths are still length $k$.

Figure 2.4: A demonstration that path-greedily searching $T_k$ only ever searches length-$k$ paths.

probes that eliminate leaves. A leaf is eliminated when we begin searching a path beginning there (because after this probe, we will have no further use for it) or when we find a path to some ancestor of that leaf (regardless of whether the current search finds a path to the root or not, the leaf will not be necessary). When all the leaves are eliminated, the tree is decided—we have either found a path to the root, or a cut that is "beneath" all the leaves.

**Lemma 2.13.** *Probing any single path in a binary tree of depth k eliminates no more than $O\left((2q)^k\right)$ leaves on average.*

*Proof.* Consider Figure 2.4. Suppose without loss of generality that we search the path starting at leaf 1.

Obviously, we always eliminate leaf 1. If the first edge is free; leaf 2 is eliminated as well. If the second edge is free *also*, leaves 3 and 4 are eliminated, and so on. In fact, it's easy to see that the number $N$ of leaves eliminated is:

$$N = 1 + \sum_{i=0}^{k-1} \left(q^k 2^k\right) = O\left((2q)^k\right)$$

□

*Remark* 2.14. This applies to binary trees that are *not* complete; in such trees, a path probe can "eliminate" leaves that never existed, but doesn't get any additional help.

Remember that there are $2^k$ leaves in a complete binary tree to start with. Since each step (probe of a single path) can only eliminate $(2q)^k$ leaves, the number of steps (and thus the search cost) is at least

$$\frac{2^k}{(2q)^k} = \Omega\left(\left(\frac{1}{q}\right)^k\right)$$

as desired. □

Lemma 2.8, Lemma 2.9, and Lemma 2.12 combine nicely to form the desired result: there exists a tree of depth $k/d$, and searching it must take $\Omega\left(\left(\frac{1}{q}\right)^{k/d}\right)$ time. □

Thus, we have demonstrate that for an important family of graphs, local search is fundamentally inefficient. We would like to show that the DFS algorithm is in fact optimal; this is difficult, but we strongly conjecture it to be true. For one thing, the bounds on any algorithm are roughly derived from the cost of searching either the initial or the final tree, and in both cases the cost of searching a full tree matches the cost of DFS; our lower bounds simply cannot guarantee a full tree. We believe that with additional work, the proportion of the tree that we can guarantee will grow; if we can eventually demonstrate that (up to constant factors) the whole tree is of interest, DFS is immediately optimal.

In fact, we can already combine the techniques used above to form a better bound for both the blocking and free regimes.

**Lemma 2.15.** *For the exponential base lower bound c given in Theorem 2.3, we have:*

$$c \geq (2q)^{-\log(q)}$$

*Sketch of proof.* Recall by Lemma 2.8 that the expected value of the number of available leaves of the converging tree of $F_k$ is $q(2q)^k$. By Lemma 2.12, if we give away the diverging tree, we then must eliminate all available leaves via path-probes into the converging tree. Furthermore, if we are probing into a tree that contains a rooted copy of a complete binary tree of depth $d$, each path-probe can eliminate at most $(2q)^d$ leaves on average (Lemma 2.13.)

But consider the pebble number of the available portion of the converging tree. It is at *most* $k \log(2q)$ in the average case, since a complete tree of higher depth would have more leaves. Thus we can eliminate on average at most $(2q)^{k \log(2q)}$ leaves on average with each path-probe, by the above lemmata. Thus gives a trivial lower bound on the number $N$ of required path-probes,

$$N \geq \frac{(2q)^k}{(2q)^{k \log(2q)}} = (2q)^{k(1-\log(2q))} = (2q)^{-k \log(q)} = \left((2q)^{-\log(q)}\right)^k,$$

as desired. □

*Remark* 2.16. This bound does not yet match that of DFS ($2q$ in the blocking range, $1/q$ in the blocking range) but it is a much more encouraging result, easily seen in Figure 2.5. It is considerably higher than our earlier bounds (in fact, in the passing regime, our earlier bounds were nonconstructive and experimentally the derived base was very close to 1). Also, the previous bounds for both the passing regime and the blocking regime approached 1 at the critical value, whereas DFS's cost is at a maximum there. This new bound is also maximal at the critical value. Lastly, as can be seen from the graph, the lower bound approaches the upper bound near $q = \frac{1}{2}$ and $q = 1$, as the bounds are tangent there. Thus we feel that even if this bound is not tight with DFS, it provides substantial supporting evidence towards a conjecture that DFS is optimal for local search of the fully-parallel graph.

DFS is still not optimal in the entire range, but we are getting closer to that goal. □

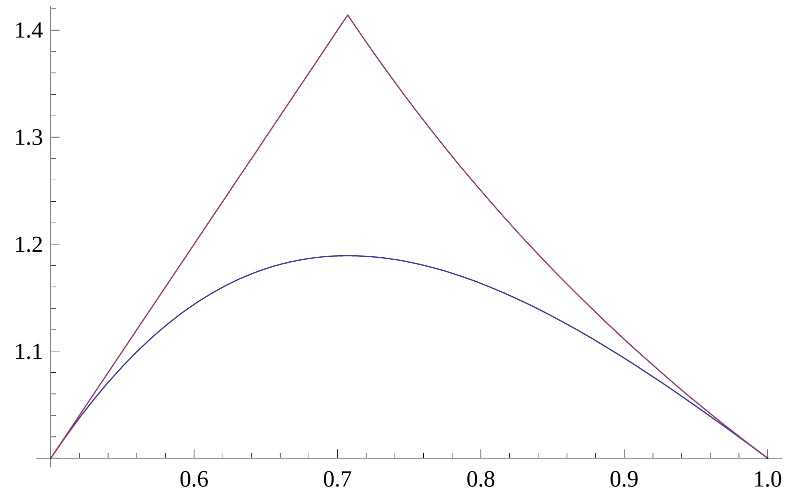


Figure 2.5: Our best upper and lower bounds for $c$, the exponential base of searching $F_k$.

# Chapter 3

# Computational Complexity of Path Search

As discussed in Section 1.4, it is a cause for some concern that we ignore the computational cost of making choices (as opposed to the cost of making probes.) The best we can say easily about the computational effort needed to find the best choice at any step is that PSPACE suffices. To the complexity theorist, this is not a reassuring bound of tractability!

We thus seek to accurately characterize the hardness of the tasks we need to perform to actually *execute* our optimal algorithms. The following are our main results:

- *PATHPROB* is #P-complete.
- *PATHCOST* is #P-hard, and contained in PSPACE.

While we have yet to define the above languages, the names should be suggestive, and the hardness results (at least) should be disheartening, at least if we are looking for a truly *efficient* (or even tractable) method of searching graphs.[1]

All the results in this chapter (unless otherwise specified) apply to *local* search algorithms. We can begin by defining the above languages.

## 3.1 Our Languages

There are two (computational) problems of interest in path search.

[1]Of course, from the perspective of a complexity theorists, such hardness results are only a good thing.

- Computing the probability some channel graph $G$ is free.
- Computing the *cost* of searching some $G$.

From here we derive our first two important problems, which we present them here as decision problems; we could easily consider them function problems as well (and in fact, will do so later for convenience).

$$\begin{aligned} PATHPROB &= \{< G, q, b > | P_F(G, q) \geq b\} \\ PATHCOST &= \{< G, S, q, b > | E(G, q) \geq b\} \end{aligned}$$

Here, as before, $S$ is the set of "accessible" vertices to begin with (under a local search assumption.) This allows us to subsume local and glocal search. For a fully local search, $S = \{s\}$; for a fully global search, $S = V$. Obviously, we do not need a starting set $S$ for the free-probability language—the set of accessible starting points does not affect the overall probability that the graph is free, only how many probes it takes to ascertain that!

## 3.2 Hardness Results for Our Languages

Our first theorem is a well-known result of Valiant.

**Theorem 3.1** (Valiant)**.** *PATHPROB is #P-complete.*

*Proof.* See Valiant (1979b). He considers the related problem of counting the subgraphs of $G$ that are connected, but a little thought shows that for $q = \frac{1}{2}$ all such subgraphs are equally likely, and the desired probability is just the number of connected graphs over the total number of subgraphs. Simple constructions extend this to other $q$. □

Theorem 3.1 will be our main building block. We begin by using it to examine computing costs.

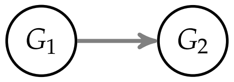
**Theorem 3.2.** *PATHCOST is #P-hard, and contained in PSPACE.*

*Proof.* For the containment, just remember that as discussed in Section 1.4 $PATHCOST$ is a game against nature and thus in PSPACE (Papadimitriou, 1985). For the hardness result, we reduce from $PATHPROB$.

Following Arora and Barak (2009) and Valiant (1979a), we will call a function problem $y$ #P-hard if given an oracle for $y$, we can compute any function in #P; that is,

$$\#P \subset FP^y.$$

In our case, as we reduce from $PATHPROB$, we will call $PATHCOST$ #P-hard if given an oracle for $PATHCOST$, we can compute $PATHPROB$ in polynomial time. Take an arbitrary instance of $PATHPROB$; as a function problem, that instance is $< G, q >$, and asks us to compute $P_F(G, q)$. Construct the graph $GG$, which is just two copies of $G$ in series.



We do not want any edges separating $G_1$ and $G_2$; instead, we simply merge the source of $G_2$ with the previous graph's target. Now, for any graph $G$ define $E_B(G, q)$ as the expected cost of the optimal algorithm[2] conditioned on $G$ being blocked; define $E_F$ similarly for the case where $G$ is free. Clearly

$$E(G, q) = P_B(G, q)E_B(G, q) + P_F(G, q)E_F(G, q), \tag{3.1}$$

but note that (in terms of the optimal algorithm for $G$) the optimal algorithm for locally searching $GG$ is obvious.

1. Search $G_1$. If it is blocked, return that $GG$ is blocked.
2. Otherwise, search $G_2$ and return the answer that gives.

In fact, no other algorithim is *possible*. If we are doing a fully local search, we must decide if $G_1$ is free or blocked before we can even consider examining $G_2$. We can see that

$$E(GG, q) = P_B(G, q)E_B(G, q) + P_F(G, q)\left(E_F(G, q) + E(G, q)\right),$$

but subtracting out Equation 3.1 and solving for $P_F(G)$,

$$P_F(G, q) = \frac{E(GG, q) - E(G, q)}{E(G, q)}.$$

Thus (labeling the sources of $G$ and $GG$ $s_G$ and $s_{GG}$), and given access to an oracle $F$ for $PATHCOST$, we can solve $PATHPROB$ in polynomial time.

1. Construct $GG$.
2. Query the oracle for $a = F(< G, \{s_G\}, q >)$.
3. Query the oracle for $b = F(< G, \{s_{GG}\}, q?)$.

[2]To be clear, this is the optimal algorithm *overall*; if there is an algorithm that is more efficient just in the case of a blocked graph, we do not use it.

4. Return the value
$$c = \frac{b-a}{a}$$

Thus with two queries to a $PATHCOST$ oracle, we can solve $PATHPROB$, and $PATHCOST$ is #P-hard. □

*Remark* 3.3. A logical extension from here is to reduce the gap between upper and lower bounds for $PATHCOST$. Toda's theorem (Arora and Barak, 2009) tells us
$$PH \subset P^{\#P}$$
and in some sense, $PSPACE$ is the "next step up" from the polynomial hierarchy. Since we know the hardness of our problem lies somewhere in this narrow range, it's natural to want to pin down its precise location. In finding a cheaper algorithm, the biggest difficulty is simply *representing* the algorithm(s) we wish to evaluate; short of a polynomially-deep and exponentially wide tree, it is simply difficult to find a way to compute with a description of an algorithm (so that we can find its cost, and then possibly find some sort of minimum.)

In the other direction (proving a PSPACE-hardness result, instead of solving $PATHCOST$ in fewer resources than unrestrictured polynomial space) the main difficulty is overcoming the local[3] action of path search. Most of the $PSPACE$-complete games against nature allow a decision made in one portion of the game to drastically and irrevocably modify other disjoint points—the proof that the dynamic graph reliability problem considered in Section 1.4 is PSPACE-complete relies critically on the ability to set up a problem where moving to one vertex of the graph can cause a failure in a completely different part of the graph.

Path search, on the other hand, has only local effects. Probing an edge can only change the status of adjacent vertices—the result cannot effect other edges, except possibly by making probing those edges unnecessary. Thus, it is difficult to apply reductions from the literature, as we cannot replicate the necessary effects.

[3]This sense of the word is completely orthogonal to *local path search*; we're simply noting the local effects of any of our decisions.

# Chapter 4

# Future Work and Conclusions

In this work, we have made the first known examination of local path search, and found many surprising results. However, much is left to be studied; many questions are still open, and there are some tempting conjectures we have yet to successfully prove. We will present some such questions and conjectures here.

## 4.1 Future Work

**Conjecture 1.** *A depth-first approach is optimal for local search on any graph.*

*Remark* 4.1. We have found no counterexample to this conjecture. In fact, for every graph we have considered, it seems that the following heuristic is true: if it's optimal to make your next probe in some subgraph of $G$, unless that probe determines the free/blocked status of that subgraph, it's still optimal to keep probing there.

The intuition behind this is simple. If we expect to make the most "progress" by looking at one part of $G$, it's probably because we can find a path there (or determine that some large part of $G$ is blocked). It's unlikely that some result of that probe will tell us suddenly that there's nothing that can be easily learned from that section of $G$.

If we could prove this greedy property of path search, the conjecture would immiediately follow (for a wide class of graph families, at least.) Methods for doing so are unclear.

**Conjecture 2.** *If indegree and outdegree are bounded by a constant for some family $G_k$, and the family grows exponentially in size and polynomially in depth with $k$, then the family cannot be locally searched in polynomial time. In other words,* *there are* no *efficiently searchable families (in the sense of Section 1.2.3) in the local case.*

*Remark* 4.2. This conjecture is derived from Conjecture 1 (though not necessarily dependent on it.) Our intuition is simple: any graph meeting the characteristics has to branch out from the source, repeatedly and often, or it can't have enough vertices. While we don't know, obviously, if this arbitrary family's branching is a tree, it shares a lot of characteristics with the trees of $F_k$—and similar arguments would hopefully prove we must explore nearly all of it to find a cut or path.

**Conjecture 3.** *$PATHCOST$ is $PSPACE$-complete.*

*Remark* 4.3. As mentioned above, this seems true, as it certainly seems difficult to calculate algorithm costs without the ability to delve into a polynomially-deep tree, but is hard to prove without nonlocal effects most reductions rely upon.

## 4.2 Conclusions

As is often true, restricting the options available in the path search problem has produced a new, interesting problem with some structure not known to exist in global path search—several tractable graphs become intractable, common techniques for finding efficient search algorithms become inapplicable, and we find several previously-unknown hardness results (that do not immediately apply to the global case.) Much is still to be done, the topic is accessible, and the local case is at least nominally more connected to the practical application/justification of routing networks. Given these encouraging facts, we look forward to seeing what else can be said about local path search.

# Bibliography

Arora, Sanjeev, and Boaz Barak. 2009. *Computation Complexity: A Modern Approach*. New York, NY, USA: Cambridge University Press. URL `http://www.cs.princeton.edu/theory/complexity`. 1.2.1, 1.4, 3.2, 3.3

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. *Introduction to Algorithms*. Cambridge, MA, USA: MIT Press. 1.2.1

Flajolet, Philippe, Jean-Claude Raoult, and Jean Vuillemin. 1979. The number of registers required for evaluating arithmetic expressions. *Theoretical Computer Science* 9:99–125. 2.3

Iwata, Shigeki, and Takumi Kasai. 1994. The othello game on an $n \times n$ board is pspace-complete. *Theoretical Computer Science* 123(2):329–340. 1.4

Lin, Geng, and Nicholas Pippenger. 1996. Routing algorithms for switching networks with probabilistic traffic. *Networks* 28(1):21–29. 2.1, 2.2, 2.2

Papadimitriou, Christos H. 1985. Games against nature. *Journal of Computer and System Sciences* 31(2):288–301. 1, 1.4, 3.2

Pippenger, Nicholas. 1999. Upper and lower bounds for the average-case complexity of path search. *Networks* 33(4):249–259. 2.2

Valiant, L. G. 1979a. The complexity of computing the permanent. *Theoretical Computer Science* 8(2):189–201. 3.2

Valiant, Leslie G. 1979b. The complexity of enumeration and reliability problems. *SIAM Journal on Computing* 8(3):410–421. 3.2