

Solving Graph Connectivity Problems on JAGs

by

Parry Husbands

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1994

© Massachusetts Institute of Technology 1994. All rights reserved.

Author

Department of Electrical Engineering and Computer Science

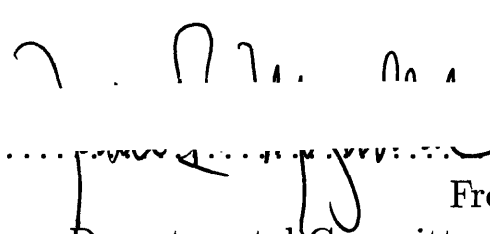
May 26, 1994

Certified by MAY 26, 94

Mauricio Karchmer

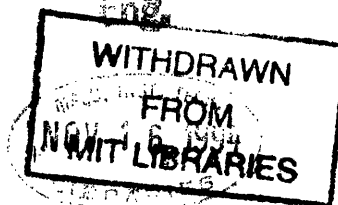
Assistant Professor of Applied Mathematics

Thesis Supervisor

Accepted by


Frederic R. Morgenthaler

Chairman, Departmental Committee on Graduate Students



Solving Graph Connectivity Problems on JAGs

by

Parry Husbands

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 1994, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

The Jumping Automaton for Graphs (JAG) is a model of computation introduced by Cook and Rackoff to study the computational complexity of st -connectivity. Although it is more restricted than a general Turing Machine, it can implement most known algorithms for this and related problems. Therefore, a fruitful intermediate step in understanding st -connectivity to attempt to prove tight upper and lower bounds on this model.

We will discuss variants of the model and give examples that demonstrate both the power of JAGs and also the difficulties involved in proving lower bounds. Space upper bounds by Cook and Rackoff will also be presented as well as space lower bounds by Poon that show how these difficulties can be overcome. Time lower bounds and time-space tradeoffs will also be discussed.

Thesis Supervisor: Mauricio Karchmer

Title: Assistant Professor of Applied Mathematics

Acknowledgments

I would like to thank my advisor Mauricio Karchmer for his guidance and patience during the writing of this work. I'm also grateful for the advice and help that my graduate counselor Albert Meyer has provided over the last two years. Finally, many thanks to my parents Parry and Genetha Husbands and my sister Mary-Gene for their love, support, and encouragement.

Contents

1	Introduction	7
1.1	Graph Connectivity	7
1.2	JAGs	7
1.3	Universal Traversal Sequences	10
1.4	Overview of Results	11
2	An example	12
2.1	NJAGs	12
2.2	The Graphs	12
2.2.1	Skinny Trees	12
2.2.2	Modified Skinny Trees	13
2.3	The Lower Bound	15
2.4	An Upper Bound	15
2.5	Interaction	16
3	An Upper Bound	17
3.1	Overview	17
3.2	The Details	18
3.2.1	The Check	19
3.2.2	Space Analysis	22
4	A Lower Bound	23
4.1	Introduction	23
4.2	Skinny Trees Revisited	23

4.3	The Lower Bound	24
4.3.1	Dealing with Interaction	24
4.4	The Main Lemma	26
4.4.1	Base Case - no interaction	27
4.4.2	The Induction Step	32
4.5	Proof of Theorem 1	37
5	Time-space tradeoffs	39
5.1	Introduction	39
5.2	A tradeoff for JAGs	39
5.2.1	The graphs	40
5.2.2	The proof	41
5.3	A tradeoff for modified NJAGs	44
5.3.1	The graphs	45
5.3.2	The proof	46
6	Conclusion	50
6.1	Other Results	50
6.1.1	Probabilistic JAGs	50
6.1.2	Tradeoffs	51
6.1.3	Directed <i>st</i> -nonconnectivity	51
6.1.4	Undirected Graphs	51
6.2	The future	51

List of Figures

1-1	A JAG	8
1-2	The positions of the pebbles after the move “Walk pebble 4 along edge 2”	9
1-3	NO-JAG and NN-JAG Pebble Information	10
2-1	The nodes of a skinny tree	13
2-2	A sample skinny tree	13
2-3	$MS(01)$	14
3-1	The positions of the pebbles before the check is started	19
3-2	The positions of the pebbles when J_4 is started	20
3-3	Flow of control in the algorithm (inner boxes denote important tests)	21
4-1	A generalized skinny tree	25
5-1	A layered graph	40
5-2	The transformation we hope to achieve	43
5-3	A squirrel cage graph with 16 nodes	45
5-4	A sample input graph (the dashed edges have been switched)	46

Chapter 1

Introduction

1.1 Graph Connectivity

An important open question in Complexity Theory is the \mathcal{L} vs. \mathcal{NL} question – are there languages that can be decided by nondeterministic logspace Turing Machines that need superlogarithmic space on deterministic Turing Machines? As with the \mathcal{P} vs. \mathcal{NP} problem we have the notion of completeness for \mathcal{NL} with respect to logspace reductions. The set $\{ \langle G, s, t \rangle : s \text{ and } t \text{ are connected in directed graph } G \}$ is complete for \mathcal{NL} and so pinning down the space complexity of st -connectivity will settle the above question. Savitch’s Theorem tells us that $\mathcal{NL} \subseteq \mathcal{DSPACE}(\log^2 n)$ and theoreticians have tried for many years to either improve the simulation or prove a good lower bound on the complexity of st -connectivity.

1.2 JAGs

Most popular algorithms for st -connectivity, breadth and depth-first search, for example, mark previously visited nodes (s is initially marked) and then explore new areas of the graph by marking neighbours of these nodes. JAGs, introduced by Cook and Rackoff [4], formally capture this strategy.

A JAG (Jumping Automaton for Graphs) is a Turing Machine that accesses its input graph in a restricted way. It has no tapes, and “reads” the graph using a set

of objects called pebbles. To start off the computation, they are placed on the nodes of the graph in some initial configuration. Based the coincidence partition of the pebbles (which pebbles are on the same nodes) and the current state, the transition function specifies the new state and next move. JAG moves are of two types – walks and jumps. In a walk, a pebble is moved along an outgoing edge (the edges are labelled), and in a jump a pebble is moved to the node occupied by another pebble. An example of a walk is “Walk pebble 4 along edge 2” and an example of a jump is “Jump pebble 1 to pebble 6”.

Intuitively, a JAG can be thought of as a finite state machine giving instructions to a “black box” which contains the graph and the pebbles. After every move, the box returns the coincidence partition of the pebbles, so that the next move can be computed. See Figures 1-1 and 1-2 for an example of a JAG in action.

Notation. In what follows we will always use P to refer to the set of pebbles and Q to refer to the set of states.

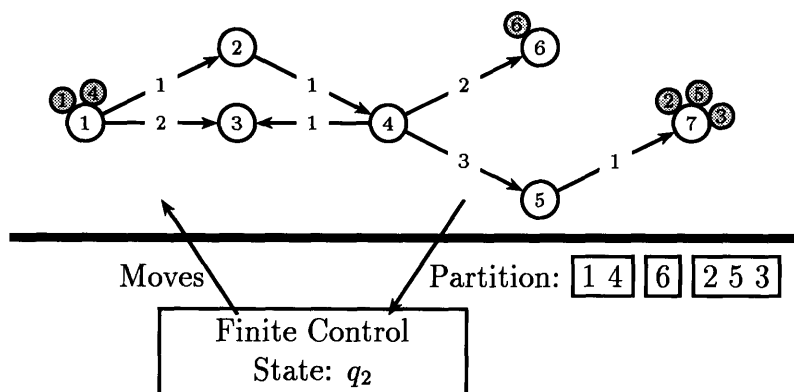


Figure 1-1: A JAG

For solving st -connectivity problems, the JAG is started out with $|P| - 1$ pebbles on s and the remaining pebble on t and it must accept its input graph if and only if s is connected to t .

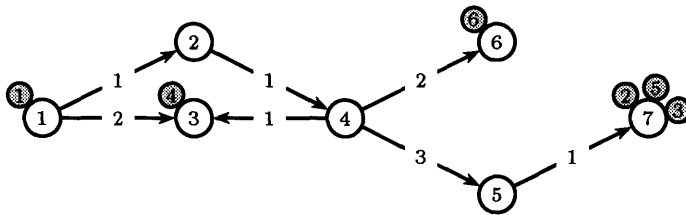


Figure 1-2: The positions of the pebbles after the move “Walk pebble 4 along edge 2”

JAGs, like boolean circuits, are non-uniform models of computation. For every n and d , we are allowed to give a JAG that works correctly on all n node, d degree graphs. The complexity measures that are important here are time and space. The time taken is simply the number of moves made, and the space used is $|P| \log n + \log |Q|$. Basically, we need this much space to simulate a JAG by a general Turing Machine – $\log n$ bits for the position of each pebble, and $\log |Q|$ bits to keep track of the state.

Note that a JAG only knows the partition of the pebbles – that pebbles 2, 3, and 5 are on the same node, for example. It cannot directly tell that they are on node 7 or that they are not on node 1. To overcome this limitation, Poon [6] improved the basic model in two ways.

In a Node-Ordered JAG (NO-JAG), the finite control knows not only the partition, but the ordering of the classes in the partition. We order the classes by saying that $P_1 < P_2$ if the label of the node that contains pebbles P_1 is less than that of the node containing P_2 . In a Node-Named JAG (NN-JAG) the finite control knows the mapping from pebbles to nodes. Figure 1-3 shows the information that an NO-JAG and an NN-JAG would receive for the graph shown in Figure 1-1.

In reasoning about JAGs, we need to be able to refer to a snapshot of a JAG’s computation much like a configuration of a Turing Machine. For a computation on a graph, an instantaneous description (ID) consists of the current state and the partition of the pebbles. For NO-JAGs and NN-JAGs, we include the additional information.

NO-JAG: $\boxed{1\ 4} < \boxed{6} < \boxed{2\ 5\ 3}$

	Pebble	Node
NN-JAG:	1	1
	2	7
	3	7
	4	1
	5	7
	6	6

Figure 1-3: NO-JAG and NN-JAG Pebble Information

1.3 Universal Traversal Sequences

One way to see that JAGs are able to solve st -connectivity on undirected and strongly connected directed graphs is to observe that 1 pebble JAGs can follow Universal Traversal Sequences.

Definition. For graphs of degree d , a Universal Traversal Sequence for n is a sequence of edge labels (i.e. integers from 1 to d) that, if given any n node graph, any labelling of the edges and any starting node, the path in the graph defined by following each outgoing edge in the sequence touches every node. For graphs that are not regular, the edge numbers range from 1 to n .

For strongly connected directed graphs, it is not too difficult to show that such sequences exist and must have exponential length. Thus, if the moves to be made are “hard coded” into the finite control, we have a 1 pebble JAG for st -connectivity on strongly connected directed graphs that uses polynomial space. For undirected graphs, however, better bounds can be obtained. A survey of these results can be found in [7]. Some of them that can be applied to JAGs follow:

- Undirected Universal Traversal Sequences of polynomial length exist. Thus, a 1 pebble JAG can solve undirected st -connectivity using $O(\log n)$ space and polynomial time.

- It is possible (for a Turing Machine) to construct undirected Universal Traversal Sequences using $O(\log^2 n)$ space.

1.4 Overview of Results

We start our tour of JAG results with an example that proves that, on a wide class of graphs, a nondeterministic JAG with one pebble needs an exponential number of states to correctly solve directed st -nonconnectivity. This example demonstrates some techniques that are used in proving lower bounds and also shows that when given more pebbles, JAGs can drastically reduce the number of states needed.

Before proceeding any further, we provide some evidence that JAGs are reasonable models, by presenting an algorithm by Cook and Rackoff [4] that solves directed st -connectivity using $O(\log^2 n)$ space – the upper bound proved by Savitch for Turing Machines.

A lower bound by Poon [6] is then given, that proves that NN-JAGs require $\Omega(\frac{\log^2 n}{\log \log n})$ space to solve directed st -connectivity.

Time-space tradeoffs are then discussed. We describe a result by Barnes and Edmonds [1] who prove that the time-space product for directed st -connectivity on JAGs is $\Omega(\frac{n^2}{|P| \log^2(n/|P|)})$. Finally, we give a proof by Beame, Borodin, Raghavan, Ruzzo, and Tompa [2] that shows that a modified nondeterministic JAG with some unmovable pebbles needs time $\Omega(n^2/|P|)$ to solve undirected st -nonconnectivity.

Chapter 2

An example

2.1 NJAGs

In this chapter we will examine the space complexity of solving st -nonconnectivity on 1-pebble nondeterministic JAGs (NJAGs). Nondeterminism is introduced to the model in the usual way – for every ID the NJAG can choose the next state and move from a set of possible candidates. We will add the additional constraint that all branches of the nondeterministic computation must end, i.e. we can't get into infinite loops.

The main result of this chapter shows that 1-pebble NJAGs need superlogarithmic space to solve st -nonconnectivity. Like most lower bounds on this model, we prove this by coming up with a “difficult” family of graphs and showing the lower bound when inputs are restricted to this family.

2.2 The Graphs

2.2.1 Skinny Trees

Skinny trees were invented by Poon [6] for proving lower bounds on NN-JAGs. For every k , there exists a family of trees with $2k + 3$ nodes. Each tree consists of a root (s) and $k + 1$ levels containing 2 nodes each. Each level has a left side node (l_i) and

a right side node (r_i). See Figure 2-1 below:

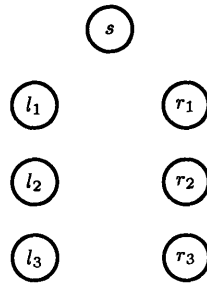


Figure 2-1: The nodes of a skinny tree

To produce the set of trees we vary the edges. For every $x \in \{0, 1\}^k$ there is a different skinny tree $\mathcal{S}(x)$. It is constructed as follows:

- All trees have the edges (s, l_1) and (s, r_1) .
- If x_i (the i th bit of x) is zero, then we include the edges (l_i, l_{i+1}) and (l_i, r_{i+1}) . If not, then we include (r_i, l_{i+1}) and (r_i, r_{i+1}) .

Figure 2-2 shows a skinny tree with $k = 2$ and $x = 01$.

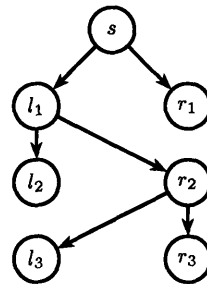


Figure 2-2: A sample skinny tree

2.2.2 Modified Skinny Trees

For the purposes of the lower bound of this chapter, some more nodes are added to the trees. This is done so that all leaves are on the same level. We add two new nodes (bl_i and br_i) to each level except the first. For $i = 2 \dots k$ the following edges are added:

- Edges $(bl_i, bl_{i+1}), (bl_i, br_{i+1}), (br_i, br_{i+1}),$ and (br_i, bl_{i+1})
- If x_{i-1} is zero, edges (r_{i-1}, bl_i) and (r_{i-1}, br_i) . If not we add edges (l_{i-1}, bl_i) and (l_{i-1}, br_i)

As before, for each $x \in \{0, 1\}^k$ there is a Modified Skinny tree $\mathcal{MS}(x)$ with $4k + 3$ nodes.

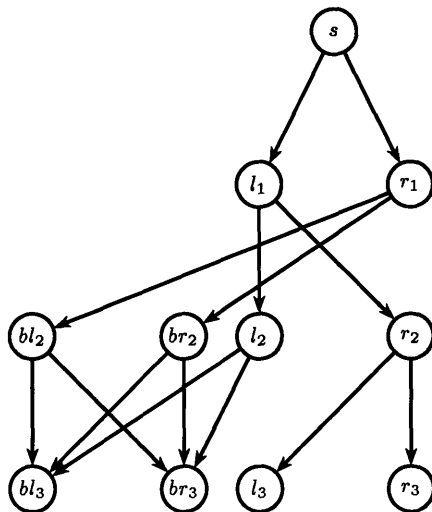


Figure 2-3: $\mathcal{MS}(01)$

If a pebble is ever on a node with label bl_i or br_i , we say that it is on the bad side. Otherwise it is on the good side.

The graphs used in the lower bound below will be modified skinny trees with an additional isolated vertex t . Since the NJAG only has 1 pebble, we provide two fixed pebbles, one on s and one on t . They can't ever be moved but they help the NJAG tell when s and t are visited. We start off the computation with the movable pebble on s .

Before proving the lower bound, we first argue that a 1-pebble NJAG can solve st -nonconnectivity on this family of graphs. There are $O(2^{k+1})$ paths from s to the leaves. Each path can be thought of as a $k + 1$ -bit string indicating, at each node in the path, whether the first or second edge is taken. Therefore, with $O((k + 1)2^{k+1})$ states the NJAG can deterministically follow each of those paths and verify that there

is no way to get to t . We can have one state for each of the $k + 1$ moves made down each of the 2^{k+1} different paths.

2.3 The Lower Bound

Theorem 1. 1-pebble NJAGs (with fixed pebbles on s and t) need superlogarithmic space to solve st -nonconnectivity.

Proof: The proof rests on the key observation that any accepting branch for $\mathcal{MS}(x)$ is an accepting branch for all $y \in \{0,1\}^k$. To see this note that the coincidence partition of the NJAG doesn't depend on x . Note also that since we added the bad side, the degree of the node that the pebble is on at any time also doesn't depend on x . It depends only on the number of walks from s .

Therefore any accepting branch must work on all 2^k modified skinny trees. On any accepting branch both good side leaves must be visited, else putting an edge from the leaf that isn't visited to t will result in a graph that is st -connected but is accepted by the NJAG. The only way to visit these leaves is to walk from s . But any walk from s visits a good leaf of only 1 modified skinny tree. So we must have at least 2^k walks from s in any accepting branch. Each of these walks must start in a different state or else the NJAG may loop. Thus we must have at least 2^k states. Since the graphs have $4k + 4$ nodes (including t), the space used by this NJAG must be at least $\log 2^{\frac{n-4}{4}} = \frac{n-4}{4}$. ■

2.4 An Upper Bound

It is relatively easy to see that an NJAG with 4 pebbles can solve st -nonconnectivity on this family of graphs using $O(\log n)$ space. The algorithm is based on the fact that there are many different paths to nodes bl_{k+1} and br_{k+1} . We can get to the bad side from s by following either edge leaving s .

To visit all of the leaf nodes we perform the following steps:

1. Pebble 1 guesses a path to a bad side leaf, first following edge 1 from s . We can assume here that the NJAG “knows” when it has reached a node with no outgoing edges.
2. Pebble 2 verifies that pebble 1 is indeed at a bad side leaf by guessing a path to pebble 1, but first following edge 2 from s . It will verify this if it meets up with pebble 1.
3. Pebble 2 jumps back to s , and guesses a path to the other bad side leaf.
4. Pebble 3 verifies that pebble 2 is at the other bad side leaf.
5. Pebble 3 guesses a path to a good side leaf. It will be at a good side leaf if, after $k + 1$ walks from s , it is at a leaf and hasn’t encountered pebbles 1 or 2.
6. Pebble 4 guesses a path to the other good side leaf. If there is no edge to t from any of the good side leaves, we accept the input graph.

For each step above, it takes a constant number of states to guess a path from s to a leaf and so the total space used by this NJAG is $O(\log n)$.

2.5 Interaction

It is interesting to note where the lower bound proof breaks down when dealing with 4 pebbles. The key observation no longer holds. An accepting branch for x may not be an accepting branch for y since there may be pebble collisions (pebbles meeting because of a walk) in the computation on x that don’t occur with y . It seems that pebble interaction plays a very important role in helping JAGs solve graph connectivity problems. As we can see, they also frustrate attempts to prove good lower bounds. The other lower bounds discussed in later chapters show how interactions are taken into account.

Chapter 3

An Upper Bound

3.1 Overview

As evidence that JAGs are reasonable models to study, we present in this chapter the following result of Cook and Rackoff [4]:

Theorem 2. JAGs can solve directed st -connectivity using $O(\log^2 n)$ space.

This bound matches the best known space bound of Savitch's Theorem and, in fact, the JAG algorithm can be thought of as a "bottom-up" version of Savitch's recursive Turing Machine algorithm.

Suppose we had a JAG J that could visit every node within a distance of 2^k nodes from s . With a few extra pebbles we would like to use some copies of this JAG to explore out to a distance of 2^{k+1} . One thought is to have 3 copies of J : J_1 , J_2 , and J_3 . We initially run J_1 with all pebbles starting at s . Each time J_1 thinks it has explored a new node, we jump a new pebble r to that node to mark it. We then jump all other pebbles to r and then simulate J_2 . J_2 will attempt to visit every node within a distance of 2^k from r . After it has done so, we then try to put the pebbles in the same configuration as they were when J_1 visited r so that it can continue. To do this we jump all pebbles except r to s and then run J_3 until we visit r .

This has a small problem, however. It is not entirely clear how J_1 knows when it has visited a new node for the first time. If it visits a node b more than once and J_2

is run from b a second time, J_3 would restore the pebbles to how they were when b was visited initially, and our algorithm would get into an infinite loop. With a few more copies of a modified version of J , and some more pebbles, a working algorithm can be devised.

3.2 The Details

The formal proof is by induction on k . With 1 pebble, a JAG can visit all nodes 1 step away from s by just moving the pebble along each outgoing edge and jumping back to s . The induction step shows how to build a JAG H that covers twice the distance as a JAG J :

The idea is to avoid getting into an infinite loop after J_1 's pebbles are restored. This can be ensured by requiring that the JAG J that searches to a distance of 2^k from s enter a special state q_d when it “thinks” it visits nodes for the first time. We can specify that it visits all nodes within 2^k steps from s at least once and for nodes at a distance of 2^k steps from s , it enters state q_d exactly once when each one is visited by pebble 1. Armed with this modified JAG, we can avoid the pitfalls of the previous try.

- Each time J_1 enters state q_d , we jump all pebbles including a new pebble r_1 to the node being visited by pebble 1 of J_1 . Since we can be sure that we haven't run J_2 yet from this node (since we enter state q_d only once), we start running J_2 from the node scanned by r_1 .
- Each time J_2 enters state q_d , we want to be sure that the node being visited by pebble 1 hasn't been touched before. We therefore jump another new pebble r_2 to this node and then jump all pebbles except r_1 and r_2 to s and we perform a check using more copies of J .
- If the node containing r_2 hasn't been scanned before, H enters state q_d and continues. If not, then we simply continue without going into q_d .

- When J_2 is finished exploring all nodes within a distance of 2^k steps from s , we must restore the pebbles to their places before J_2 started. Since q_d is entered at most once for each node visited, we jump all pebbles except r_1 to s and simulate J until pebbles 1 and r_1 meet in state q_d .

Assuming that we can check whether r_2 had been previously visited, we see that the above algorithm succeeds in visiting all nodes within a distance of 2^{k+1} from s and enters state q_d only once for nodes 2^{k+1} steps from s .

3.2.1 The Check

When r_2 is visited, we must decide whether to enter state q_d or not. We want to ensure that q_d is entered exactly once for nodes 2^{k+1} steps away, and at most once for closer nodes. Since we only enter q_d for nodes visited by J_2 , we can re-run J_1 and J_2 with two new JAGs (J_3 and J_4) to tell whether r_2 was visited before by J_2 in state q_d .

Figure 3-1 below shows the configuration of the pebbles before the check is started.

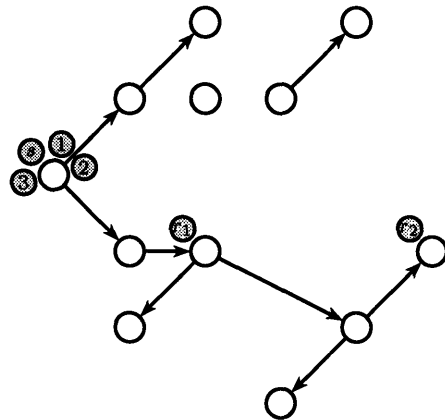


Figure 3-1: The positions of the pebbles before the check is started

- Let J_3 simulate J starting from s . If we enter state q_d and pebble 1 isn't at the same node as r_1 , we jump another new pebble r_3 to the node being scanned by pebble 1.

- Another copy J_4 then continues the search, this time starting at r_3 (see Figure 3-2). If it ever meets r_2 in state q_d , we know that it has been visited before.
- If so, then J_7 attempts to restore the pebbles for J_2 . We first jump all pebbles except pebble r_2 to r_1 and simulate J until we are in state q_d and pebble 1 is visiting r_2 .
- If J_4 never meets r_2 in state q_d , we jump all pebbles except r_1 , r_2 , and r_3 to s , and J_5 attempts to restore the pebbles for J_3 . It does so by simulating J until it reaches r_3 in state q_d .
- The check is completed when J_3 reaches r_1 in state q_d . At this point we can be sure that r_2 hasn't been visited.

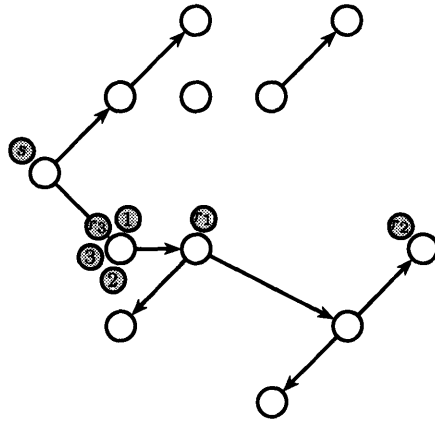


Figure 3-2: The positions of the pebbles when J_4 is started

The key point here is the use of q_d to avoid looping. One final detail needs to be addressed. H starts its search at s , but in the proof the copies of J start out their searches at any node. To fully complete the induction step we note that H can be easily modified to perform a search starting at any distinguished pebble.

Figure 3-3 gives a pictorial description of the algorithm.

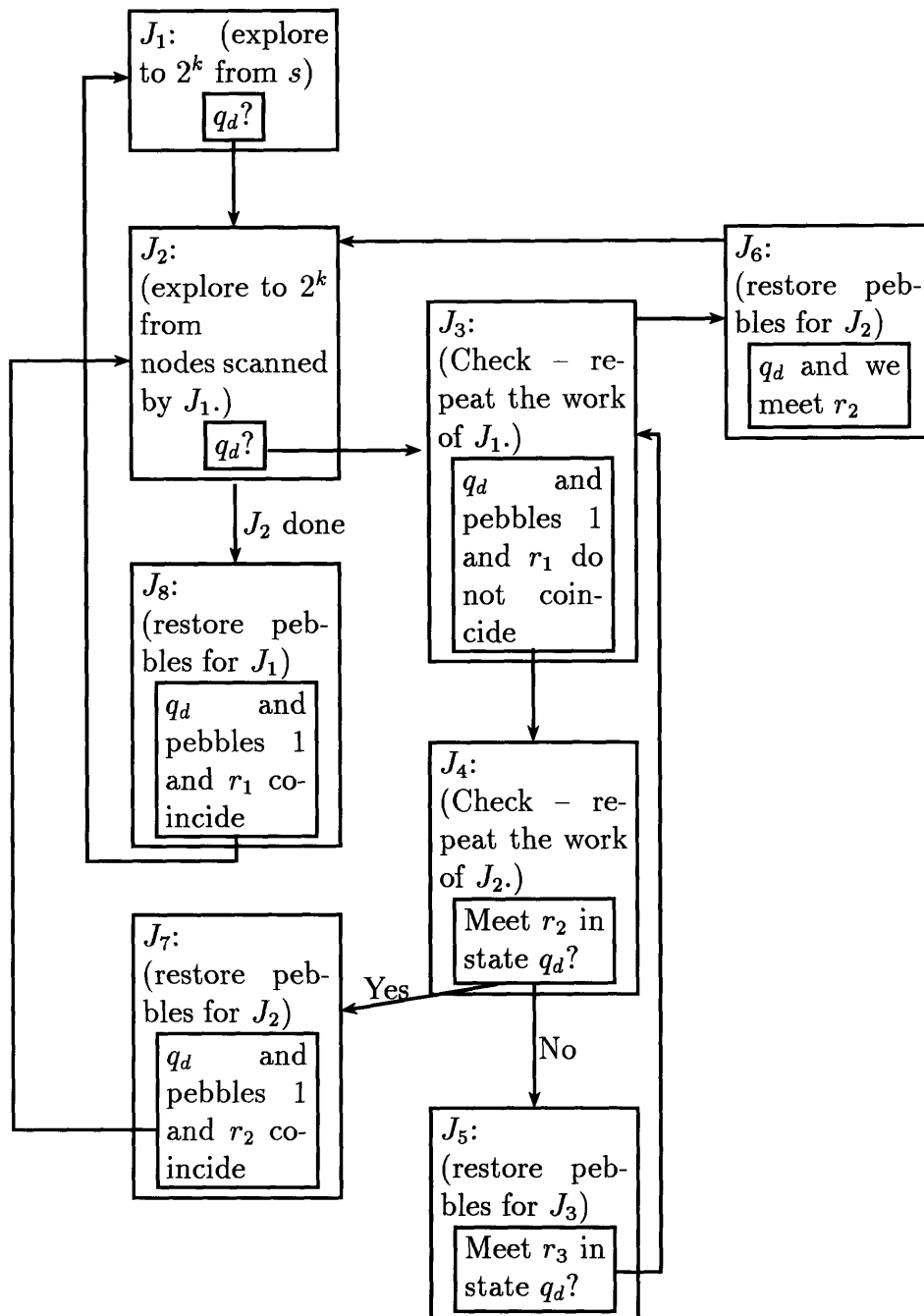


Figure 3-3: Flow of control in the algorithm (inner boxes denote important tests)

3.2.2 Space Analysis

Note that the 8 copies of J use the same pebbles to perform the simulation. Only 3 new pebbles are introduced. Thus, if $P(k)$ represents the number of pebbles used to search to a distance of 2^k from s , then $P(k+1) = P(k) + 3$ with $P(0)$ being some constant. Thus $P(k) \in O(k)$.

Let $S(k)$ represent the number of states that are used. We can easily see that $S(k+1) \leq 9S(k)$ ($8S(k)$ for the copies of J and another one for any additional states that we might need). $S(0)$ is also constant. Thus $S(k) \in O(9^k)$ and so the space used by the JAG is $O(k \log n + k)$. Since we stop this construction when $k = \log n$, we visit all nodes within a distance of $2^{\log n} = n$ from s using $O(\log^2 n)$ space.

Chapter 4

A Lower Bound

4.1 Introduction

Chapter 3 showed that Savitch's upper bound applies to JAGs. In this chapter we will prove something not known for general models of computation – that NN-JAGs require superlogarithmic space to solve st -connectivity. It is hoped that results like this shed some light on the general problem.

Cook and Rackoff [4] proved the first such result for JAGs and recently Poon [6] showed the same bound for NN-JAGs. Both results share the same basic plan – using iteration to build up a difficult graph and, at the same time, taking pebble interaction into account. Poon's proof is described below.

4.2 Skinny Trees Revisited

As always, we need a family of graphs. Skinny trees, described in Chapter 2 will be generalized as follows:

- The skinny tree $\mathcal{S}(x)$ will have a goal node that is the left leaf (l_k) if the last bit of x is 0, or the right leaf (r_k) otherwise.
- The graph $\mathcal{GS}(x, y)$ is built by replacing each node in $\mathcal{S}(y)$ with a copy of $\mathcal{S}(x)$. Some additional edges are needed to make this graph connected. If there is an

edge from node a to node b in $\mathcal{S}(y)$, there will be an edge from the goal of the copy at a to the root of the copy at b . We will denote the copy at a by C_a . The root of the new graph is the root of C_{root} and the goal is the goal of C_{goal} .

- Similarly, to get $\mathcal{GS}(x_1, x_2, x_3, \dots, x_j)$, copies of $\mathcal{GS}(x_1, x_2, x_3, \dots, x_{j-1})$ are used to replace the nodes of $\mathcal{S}(x_j)$ and additional edges are introduced from goal to root as described above.

Figure 4-1 shows $\mathcal{GS}(01, 10)$. Note that $\mathcal{GS}(x_1, x_2, x_3, \dots, x_j)$ consists of many copies of $\mathcal{S}(x_1)$.

4.3 The Lower Bound

Theorem 3. NN-JAGs require $\Omega(\log^2 n / \log \log n)$ space to solve directed st -connectivity.

4.3.1 Dealing with Interaction

Chapter 2 showed that pebble interaction can help a JAG obtain connectivity information about a graph. To this end, we need some way of capturing what goes on when pebbles meet.

Given the mapping from pebbles to nodes, we can partition the pebbles in groups called blocks in the following natural way: two pebbles can be in the same block only if they are on the same node. Note that the coincidence partition implicitly defines a set of blocks.

One way to follow pebble interaction is to see what happens to these blocks as the JAG computation progresses. At each step t , we can define the continuation of block i at step t , $B_i(t)$ as follows:

If a pebble r of block i meets, by a walk or a jump, a pebble of block j , then intuitively, the JAG “knows” something about how the portions of the graph visited by blocks i and j are connected. In this case, to obtain the blocks at time $t + 1$ we

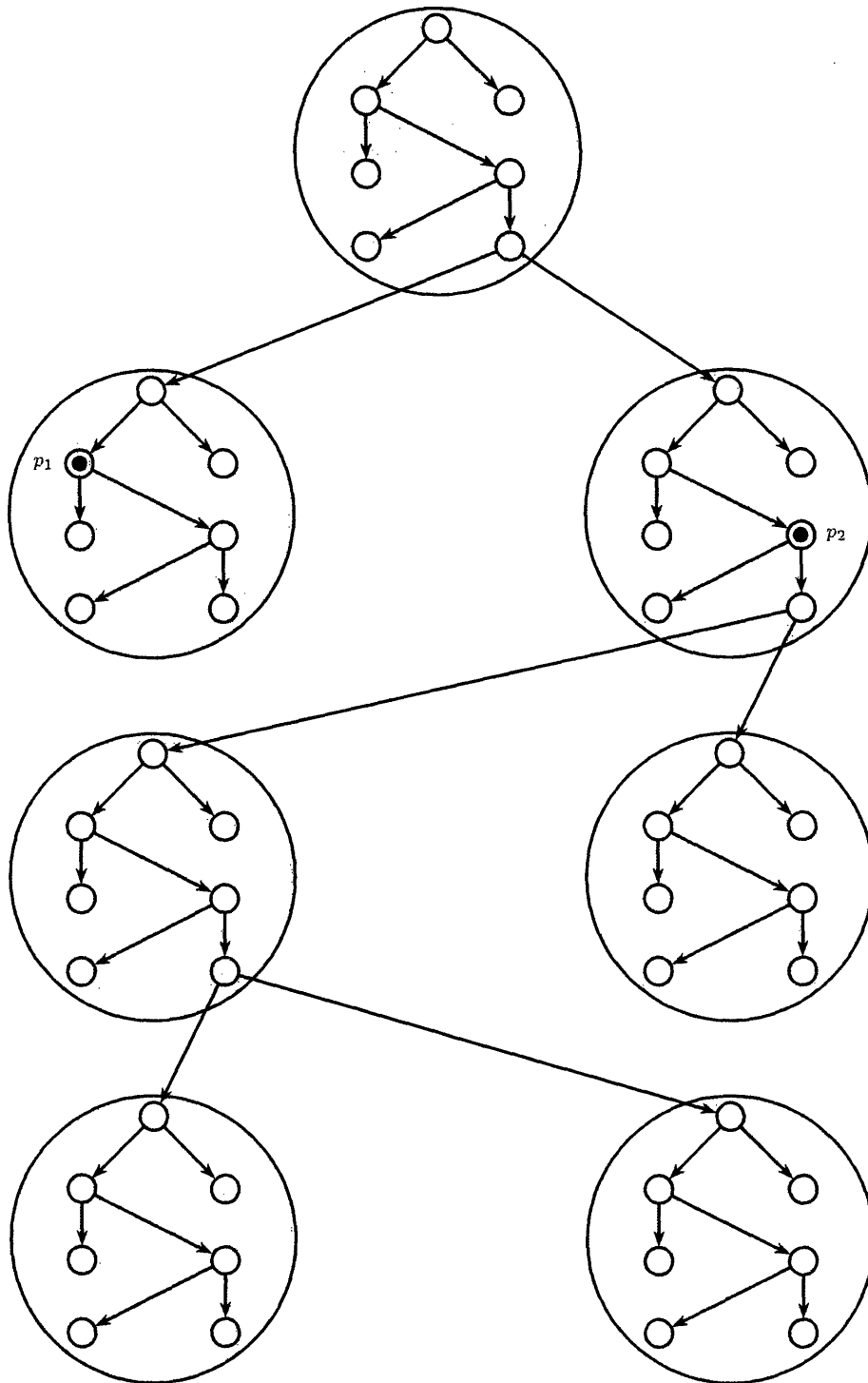


Figure 4-1: A generalized skinny tree

delete pebble r from block i and add it to block j . If moves don't cause inter-block collisions, then the blocks remain unchanged at this step.

In the proof that follows we will be trying to stop blocks of pebbles from exploring too much of the graph. The following definition will be useful:

Definition. A block B traverses a copy of $\mathcal{GS}(x_1, x_2, \dots, x_l)$ if, when the computation is started, no pebble of B is below the root of $\mathcal{GS}(x_1, x_2, \dots, x_l)$, but at a later time, some pebble of B visits its goal.

4.4 The Main Lemma

The number of non-empty blocks in the computation gives us some idea of the amount of pebble interaction that has occurred. If there are many non-empty blocks, then we can be certain that the blocks have worked somewhat independently to explore the graph. Conversely, if there are few non-empty blocks, then the JAG has learned a lot of information by way of pebble collisions. Since it is difficult to deal with all of the ways that pebbles can meet, we construct a hard graph inductively. We will construct a sequence of graphs G_1, \dots, G_p with the following properties:

- If we allow $p - k + 1$ blocks to remain non-empty, then traversing a copy of G_k is impossible.
- G_{k+1} is constructed from G_k by replacing nodes of $\mathcal{S}(x_{k+1})$ with copies of G_k . In other words, each G_k is a generalized skinny tree.

In the end, we will have G_p that is impossible to traverse as long as we have $p - p + 1 = 1$ non-empty block. But this always occurs and so we will have succeeded in constructing a difficult graph for the NN-JAG. The precise statement of the main lemma follows:

Lemma 1. Given an NN-JAG with p pebbles and Q states, and an integer k between 1 and p we can find

- d the length of the strings x_i (which only depends on p and Q)
- x_1, x_2, \dots, x_k

Such that for all x_{k+1}, \dots, x_p (ways of completing the graph) if we run the NN-JAG on $\mathcal{GS}(x_1, \dots, x_p)$ with any initial state and configuration of the pebbles no block traverses a copy of a $\mathcal{GS}(x_1, x_2, \dots, x_k)$ as long as $p - k + 1$ blocks remain empty. The initial block division is the one defined by the initial coincidence partition.

This is proved by induction on k .

4.4.1 Base Case - no interaction

Here there is no interaction. Requiring that p blocks remain non-empty implies that pebbles cannot meet at all. If pebbles collide, by a walk or a jump, we are violating the hypothesis. What needs to be shown here is that there is a base graph (remember that the Generalized Skinny Trees are nothing but copies of $\mathcal{S}(x)$) that can't be traversed without interaction.

An interesting point to note is that proving the base case involves more than showing that a 1 pebble NN-JAG can't traverse all $\mathcal{S}(x)$ s. Although pebbles are not allowed to interact, the NN-JAG can gain some information about the structure of the graph by examining the positions of all of the pebbles. For example, if pebble 1 started walking down a skinny tree $\mathcal{S}(x)$ and reached a leaf after 1 step, then the NN-JAG "knows" the first bit of x . There is probably a more direct way to prove the base case along these lines, but the proof that follows has the advantage that the induction step is similar.

Characterizing Computations with no interaction

We first need to define a computation history of an NN-JAG:

Definition. A computation history for an NN-JAG J on input graph G starting in state q_0 and initial pebble mapping P_0 is a sequence of state-mapping pairs

$(q_0, P_0), (q_1, P_1), \dots, (q_t, P_t)$ indicating at each step of the NN-JAG computation what state it's in and where the pebbles are.

We will be analyzing computation histories where there are no pebble collisions:

Definition. A 1-computation history for an NN-JAG J starting with initial state q_0 and pebble mapping P_0 on $\mathcal{GS}(x_1, x_2, \dots, x_p)$, denoted by $\text{CH}(1, q_0, P_0, x_1, x_2, \dots, x_p)$ is a computation history of J where no pebble interaction occurs and the last move of this history completes a traversal of a copy of $\mathcal{S}(x_1)$.

In what follows we will show that there is an x such that one cannot find $q_0, P_0, x_2, \dots, x_p$ to make $\text{CH}(1, q_0, P_0, x, x_2, \dots, x_p)$ a 1-computation history.

One additional definition that we will need describes the leaves of the graph:

Definition. The p -bit boolean vector $\text{leaf}_1(P_0, x_2, \dots, x_p)$ describes, for each pebble, whether the goal of the copy of the $\mathcal{S}(x)$ on which it rests is a leaf in $\mathcal{GS}(x, x_2, \dots, x_p)$. This vector depends only on P_0 and x_2, \dots, x_p . Connections between goals and roots don't depend on x . For example, in Figure 4-1, if we assume that pebbles 1 and 2 are on the nodes shown, then the leaf vector begins with 01.

It seems reasonable that the computation history of the NN-JAG depends on leaf_1 .

Lemma 2. If

$$\text{leaf}_1(P_0, x_2, \dots, x_p) = \text{leaf}_1(P_0, x'_2, \dots, x'_p)$$

then

$$\text{CH}(1, q_0, P_0, x, x_2, \dots, x_p) = \text{CH}(1, q_0, P_0, x, x'_2, \dots, x'_p)$$

.

Proof:

If this is to be false, there must be a point where the histories diverge. It isn't the first step since the initial states and pebble positions are the same. Thus there is an i such that the histories agree at time i but don't agree at time $i + 1$. To get a contradiction we need only consider what moves can be made:

- If the move is a walk within a copy of $\mathcal{S}(x)$, then the new state and pebble mapping must be the same.
- If the move is a jump, then since the pebble positions at time i are the same, they must be the same at time $i + 1$.
- If the move is a walk from the goal of an $\mathcal{S}(x)$ to the root of another $\mathcal{S}(x)$, the target of the walk is the same in both cases since the leaf₁ vectors are the same. This is true because we can be assured that the pebble that does the walk is leaving the copy of $\mathcal{S}(x)$ that it started off at. (If not, then it traversed the graph and the 1-computation history wouldn't have reached that far).

■

We can thus associate with every $(q_0, P_0, x, \text{leaf}_1)$ tuple, the 1-computation history for the NN-JAG J on any graph $\mathcal{GS}(x, x_2, \dots, x_p)$ with leaf vector leaf₁. We now consider the set of all 1-computation histories and count how many strings x occur in the tuples of this set.

Known bits, traversed bits, and active pebbles

As mentioned earlier, when one pebble successfully finds its way down a copy of $\mathcal{S}(x)$ the entire JAG knows what links to traverse. The concept of known bits, traversed bits and active pebbles attempt to capture this idea.

Definition. A bit position i of x is known if, according to P_0 , there is a pebble visiting a node of level i of a copy of $\mathcal{S}(x)$. For example, in Figure 4-1, bit position 1 is known since pebble 1 rests on level 1.

Definition. A bit position j of x is traversed, if it isn't known, but sometime in the computation, level j of a copy of $\mathcal{S}(x)$ is visited. A bit that isn't known and hasn't been traversed is called free.

If, in a 1-computation history, a pebble walks down a copy of $\mathcal{S}(x)$ and reaches a leaf that isn't a goal, it is lost since it has no way of reaching the goal of that

copy of $\mathcal{S}(x)$ (or any other since no jumps are possible – we decrease the number of non-empty blocks when we jump). A pebble is **active** if it isn't lost and can traverse a copy of $\mathcal{S}(x)$. If the pebble initially started out on the root of a copy of $\mathcal{S}(x)$ then it is active if it is on the path from the root to the goal of that subgraph. If it started out below the root, then it can possibly traverse the graphs which are rooted at the descendents of the goal of the graph it started out from. In this case, it's active if it's on the path to any of the goals of the descendents of the goal of the $\mathcal{S}(x)$ it started out on. In Figure 4-1, pebble 2 is active.

Given q_0 , P_0 , and leaf_1 we want to count the number of strings x such that $(q_0, P_0, x, \text{leaf}_1)$ gives a 1-computation history. This is accomplished by the following lemma that allows us to break up $\{0, 1\}^d$ into subsets that agree on the known bits and counting the number of “good” strings in each subset.

Lemma 3. Let $\text{count}(u, v) = \text{count}(u-1, v) + \text{count}(u-1, v-1)$ with $\text{count}(0, v) = 1$ and $\text{count}(u, 0) = 0$. Given q_0 , P_0 , and leaf_1 , and a subset S of $\{0, 1\}^d$, if

- For every string x in S , there are at most u free bits and v active pebbles at some time t during the 1-computation history described by $(q_0, P_0, x, \text{leaf}_1)$
- The known bits and the traversed bits at time t are the same for every string in S – (not just the indices, but the values as well)

Then the number of strings in S that give valid 1-computation histories $(q_0, P_0, x, \text{leaf}_1)$ is bounded by $\text{count}(u, v)$

Proof: This lemma is also proved by induction – this time on $u + v$. The base case occurs when $u = 0$ or $v = 0$:

- $v = 0$. Here there are no active pebbles at time t . Since we're in the middle of a 1-computation history we haven't traversed a copy of $\mathcal{S}(x)$ yet. But since all of the pebbles are lost, we can't ever do the traversal. Thus the number of “good” strings in S is 0. This equals $\text{count}(u, 0)$.

- $u = 0$. There are no free bits at time t . Thus all bits are either known or traversed. In this case S can have at most 1 element since all elements of S agree on the known and traversed bits. Therefore, the number of “good” strings is at most $1 = \text{count}(0, v)$.

The proof of the induction step rests on the observation that the computation histories of J on all x in S are the same up to time t . We know that the positions and values of the known and traversed bits are the same for all x in S . This means that all of the nodes that are visited in all of the computation histories must be on the levels that correspond to the known and traversed bits. Thus, during the computation history, all walks (there are no jumps in 1-computation histories) result in the same target and so the computation histories all agree.

Using this argument we can see that the histories agree until a bit that was free at time t becomes traversed at time t' . (Since all walks till then touch levels that were either known or traversed at time t). Let's say that bit i was changed to traversed. We now break up S into two sets S_0 and S_1 where $S_j = \{x \in S \mid \text{bit } i \text{ of } x \text{ is } j\}$. Note that that the pebble that reaches level i becomes lost in one of S_0 or S_1 (Either that node is a leaf in the graphs described by the members of S_0 or the members of S_1). Note also that at time t' there are $v - 1$ free bits (we just lost one). Therefore the induction hypothesis is satisfied at time t' for the sets S_0 and S_1 since the positions and patterns of the known and free bits agree in S_0 and S_1 . Thus, $|S| = |S_0| + |S_1| \leq \text{count}(u - 1, v) + \text{count}(u - 1, v - 1) = \text{count}(u, v)$.

It can be shown that $\text{count}(u, v) \leq u^v$ if $v > 1$. ■

Putting it all together

We want to prove that we can find x such that no matter what q_0, P_0 , and x_2, \dots, x_p we chose no copy of $\mathcal{S}(x)$ is traversed in $\mathcal{GS}(x, x_2, \dots, x_p)$ starting in state q_0 with pebble mapping P_0 if no pebble collisions occur. Lemma 2 allowed us to replace x_2, \dots, x_p with leaf_1 . Therefore, the tuple $(q_0, P_0, x, \text{leaf}_1)$ completely describes a 1-computation history.

Given q_0, P_0 , and leaf_1 we will now count how many strings x result in 1-computation histories. Once we know P_0 , we can find the known bits of each x in $\{0, 1\}^d$. Let there be b of them. We can therefore divide $\{0, 1\}^d$ into 2^b subsets S_1, S_2, \dots, S_{2^b} such that all the elements of each S_j agree on these bits. The above lemma then applies to each of the subsets at time 0. We know that we start out with p pebbles and $d - b$ free bits so we have at most p active pebbles. Therefore the number of “good” strings in each subset is bounded by $(d - b)^p \leq d^p$. Thus the total number of good strings is bounded by $2^b d^p \leq 2^p d^p = (2d)^p$ since $b \leq p$.

To bound the total number of good strings over all q_0, P_0 , and leaf_1 settings, we need to bound their sizes. Let Q be the number of states. Since leaf_1 is just a p -bit vector there are at most 2^p of these. To count P_0 , we first need to find the number of nodes in a generalized skinny tree. We know that each $\mathcal{S}(x)$ contains $2d + 1$ nodes and each time we iterate, we multiply the number of nodes by $2d + 1$. Therefore a generalized skinny tree has $(2d + 1)^p$ nodes. Now the pebble mapping is just a function from $\{1, \dots, p\}$ to the nodes of the skinny tree. So the number of these is $((2d + 1)^p)^p = (2d + 1)^{p^2}$. Thus the number of strings x that result in 1-computation histories is at most $(2d)^p Q 2^p (2d + 1)^{p^2} = (4d)^p Q (2d + 1)^{p^2}$. The total number of such strings is 2^d . To prove the base case, all we need to do is find d such that $2^d > (4d)^p Q (2d + 1)^{p^2}$. Choosing $d = 18p^2 \log(pQ)$ does the trick.

4.4.2 The Induction Step

The proof of the induction step is basically the same as the proof of the base case. The important step in the proof of the base case occurs in Lemma 3 where we note that when a bit becomes traversed, in some of the strings, a traversing pebble becomes lost. In the proof of the induction step we want to say the same thing for blocks. We assume that we have built up G_k that is impossible to traverse if $p - k + 1$ blocks remain non-empty. We now want to find x_{k+1} such that, if we replace the nodes of $\mathcal{S}(x_{k+1})$ by copies of G_k , we get G_{k+1} that is impossible to traverse even with a little more interaction.

To accomplish this, we can view traversing G_{k+1} as traversing $\mathcal{S}(x_{k+1})$ with the

nodes replaced by copies of G_k . This correspondence is further strengthened by Lemma 5 that lets us treat a block of pebbles on G_{k+1} as an individual pebble on $\mathcal{S}(x_{k+1})$ in the following way:

We know that if we allow $p - k + 1$ blocks to remain non-empty, a copy of G_k cannot be traversed. Lemma 5 proves that, if we allow $p - k$ blocks to remain non-empty, traversing a copy of G_k can only be accomplished if all of the block's pebbles lie in that copy of G_k . Therefore when a copy of G_k is traversed, the entire block is in that copy and so all of that block's pebbles can be viewed as a single pebble on a node of $\mathcal{S}(x_{k+1})$. The details of the proof follow.

Assuming that we have already found x_1, x_2, \dots, x_k , we need to find x_{k+1} such that for all ways of completing the graph (with x_{k+2}, \dots, x_p), and initial state and pebble configuration, no block manages to traverse a copy of $\mathcal{GS}(x_1, \dots, x_{k+1})$ as long as $p - k$ of them remain non-empty.

To this end we will find a way of characterizing computations that succeed in traversing $\mathcal{GS}(x_1, \dots, x_{k+1})$:

Definition. A $k + 1$ -computation history for an NN-JAG J starting with initial state q_0 and pebble mapping P_0 on $\mathcal{GS}(x_1, \dots, x_p)$ is much like 1-computation history. It is a computation history that maintains the invariant that $p - k$ blocks remain non-empty and ends when a block traverses a copy of $\mathcal{GS}(x_1, \dots, x_{k+1})$.

As with the base case, a p -bit boolean vector that describes the interconnections between goals and roots will be used.

Definition. The p -bit boolean vector $\text{leaf}_{k+1}(P_0, x_{k+2}, \dots, x_p)$ describes, for each pebble, whether the goal of the copy of the $\mathcal{GS}(x_1, \dots, x_{k+1})$ on which it rests is a leaf in $\mathcal{GS}(x_1, x_2, \dots, x_p)$.

We can now observe that $k + 1$ -computations depend only on the initial state, pebble configuration, x_1, \dots, x_k computed earlier, x_{k+1} , and leaf_{k+1} . In other words, if

$$\text{leaf}_{k+1}(P_0, x_{k+2}, \dots, x_p) = \text{leaf}_{k+1}(P_0, x'_{k+2}, \dots, x'_p)$$

then

$$CH(k+1, q_0, P_0, x_1, \dots, x_{k+1}, x_{k+2}, \dots, x_p) = \\ CH(k+1, q_0, P_0, x_1, \dots, x_{k+1}, x'_{k+2}, \dots, x'_p)$$

. The proof of this is the same as in the base case. We simply replace $\mathcal{S}(x)$ with $\mathcal{GS}(x_1, \dots, x_{k+1})$. Note that this implies that once we fix x_1, \dots, x_k , a $k+1$ computation history can be described by $(q_0, P_0, x_{k+1}, \text{leaf}_{k+1})$.

Known bits, traversed bits, and active blocks

The concepts that were introduced in the base case need to be generalized here. Since $\mathcal{GS}(x_1, \dots, x_{k+1})$ consists of many copies of $\mathcal{GS}(x_1, \dots, x_k)$ we can refer to the copies of $\mathcal{GS}(x_1, \dots, x_k)$, C_{l_i} and C_{r_i} that form “level” i of $\mathcal{GS}(x_1, \dots, x_{k+1})$.

In this case, a bit i is **known** if initially, some pebble rests on (any of the copies of $\mathcal{GS}(x_1, \dots, x_k)$ that form) “level” i of any $\mathcal{GS}(x_1, \dots, x_{k+1})$. Bit j becomes **traversed** if any C_{r_j} or C_{l_j} becomes traversed. As before, bits that are not known or traversed are **free**.

Since the initial division of the pebbles into blocks is the one induced by the coincidence partition, we are assured that the pebbles of a block start out at one node (not necessarily the same node for all blocks). This allows us to define **active blocks**. If all of the pebbles of block i start out at the root of a copy of $\mathcal{GS}(x_1, x_2, \dots, x_{k+1})$, then block i is active if it has some pebble on the path to the goal of that copy. Note that the path passes through many copies of $\mathcal{GS}(x_1, \dots, x_k)$. If block i initially lies below the root of some copy of $\mathcal{GS}(x_1, \dots, x_{k+1})$ then it is active if it has a pebble on the paths to the goals of the copies of $\mathcal{GS}(x_1, \dots, x_{k+1})$ that are the descendents of the initial $\mathcal{GS}(x_1, \dots, x_{k+1})$. Note that a block can become lost, much like a pebble, if all of its pebbles get stuck in C_{r_i} s or C_{l_i} s that replace non-goal leaves of $\mathcal{S}(x_{k+1})$. Once a block becomes lost, it can’t ever become active again. This is because the only way its pebbles can become “useful” again is to jump to other active blocks. But these jumps remove pebbles from the block. This can be seen as a justification for the rules that specify how blocks evolve.

As with the base case, we have the following analogue of Lemma 3 (blocks replace pebbles and $k + 1$ computation histories replace 1 computation histories):

Lemma 4. Let $\text{count}(u, v) = \text{count}(u - 1, v) + \text{count}(u - 1, v - 1)$ with $\text{count}(0, v) = 1$ and $\text{count}(u, 0) = 0$. Given q_0, P_0 , and leaf_{k+1} , and a subset S of $\{0, 1\}^d$, if

- For every string x in S , there are at most u free bits and v active blocks at some time t during the $k + 1$ -computation history described by $(q_0, P_0, x_{k+1}, \text{leaf}_{k+1})$
- The known bits and the traversed bits at time t are the same for every string in S - (not just the indices, but the values as well)

Then the number of strings in S that give valid $k+1$ -computation histories $(q_0, P_0, x_{k+1}, \text{leaf}_{k+1})$ is bounded by $\text{count}(u, v)$.

Proof: The proof of the base case of this lemma is the same. The induction step is essentially similar. We have to prove that at time t' , a block becomes lost (in order to use the induction hypothesis). A block becomes lost if all of its pebbles lie on a copy of $\mathcal{GS}(x_1, \dots, x_k)$ and there are no edges from the goal of this graph. If we can show that when a bit becomes traversed (a copy of $\mathcal{GS}(x_1, \dots, x_k)$ is traversed), all pebbles of the traversing block lie within that copy of $\mathcal{GS}(x_1, \dots, x_k)$, a block becomes lost and we can apply the same proof as in the base case. Lemma 5 below proves the above assertion and so the same count can be performed and the same value of d obtained.

■

Lemma 5. Choose x_1, \dots, x_k as in the induction hypothesis. For any x_{k+1}, \dots, x_p , if some block B finishes traversing a copy of $\mathcal{GS}(x_1, \dots, x_k)$ at time t , and there are $p - k$ non-empty blocks throughout the computation, then all pebbles of that block lie in that copy of $\mathcal{GS}(x_1, \dots, x_k)$ at that time.

Proof: This lemma intuitively states that traversing a copy of $C = \mathcal{GS}(x_1, \dots, x_k)$ is hard. Since pebbles may enter and leave B , we need to carefully deduce what $B(t)$ looks like.

The proof first finds the pebble r that completes the traversal of C and works backward to find a block of pebbles B_0 on the root of C such that $r \in B_0(t)$. Since B_0 is on the root of C , $B_0(t)$ is entirely within C . To show that all of $B(t)$ lies in C , we then prove that $B_0(t) = B(t)$.

We start by identifying the pebble r of the continuation of B that first reaches the goal of C . We then attempt to come up with a time t_0 and a block B_0 such that, if B_0 consists of all of the pebbles on the root of C at time t_0 , then $r \in B_0(t)$ (the continuation of B_0 at time t). To find B_0 , we work backwards from t . We show that for all nodes c_0, c_1, \dots, c_m on the path from the root to the goal of C , we can find a time t_i such that, if B_i consists of all of the pebbles at node c_i at time t_i , then $r \in B_i(t)$. Initially, $t_m = t$. Given that we know t_{i+1} we want to find t_i . We know that there are some pebbles on node c_{i+1} at time t_{i+1} (actually the set B_{i+1}). Therefore there must be a time when c_{i+1} fails to have any pebbles on it (since C is traversed). Let t_i be the latest such time before t_{i+1} . By the definition of t_i , we see that node c_{i+1} always has some pebble on it (not necessarily the same one) from time $t_i + 1$ to time t_{i+1} . Let B_i be the set of pebbles on c_i at this time. We need to show that $r \in B_i(t)$. By the way we chose t_i , $B_{i+1} \subseteq B_i(t_{i+1})$. To see this note that a pebble from B_i reaches c_{i+1} at time $t_i + 1$ (t_i is the latest time that c_{i+1} is unoccupied and the only way to reach c_{i+1} if it's unoccupied is to walk from c_i). Note also that all other pebbles that reach c_{i+1} from $t_i + 1$ to t_{i+1} belong to the continuation of B_i (if you move to a node that is already occupied by a pebble, you belong to that pebble's block. Since the node is continuously occupied by pebbles from time $t_i + 1$, all of these pebbles belong to the block of the first pebble that reached node c_{i+1} . That pebble came from B_i). Thus, since $B_{i+1} \subseteq B_i(t_{i+1})$ and $r \in B_{i+1}(t)$, $r \in B_i(t)$. So we can find a time t_0 with the property that, if B_0 is the set of pebbles at c_0 at time t_0 , then $r \in B_0(t)$.

The second part of the proof shows that $B_0(t) = B(t)$. Since B_0 is entirely within C , $B_0(t)$ and hence $B(t)$ are entirely within C . We first prove that $B_0 \subseteq B(t_0)$. Since all of the pebbles of B_0 are on one node, if some pebble of B_0 is in $B(t_0)$, then all of B_0 is in $B(t_0)$. If $B_0 \not\subseteq B(t_0)$, then B_0 and $B(t_0)$ have no pebbles in common. If this

is the case, then $B_0(t)$ and $B(t)$ have no pebbles in common. But r is in both $B_0(t)$ and $B(t)$. Thus $B_0 \subseteq B(t_0)$.

We now consider the pebble mapping P_0 at time t_0 . It divides the pebbles into a set of blocks, B_0 being one. Let n_0 be the number of these blocks that are non-empty at time t and let n be the number of non-empty blocks of the original computation at time t . We know that $n \geq p - k$ (one of the hypotheses of this lemma).

The final step of this proof shows that if B' is a block defined by P_0 , and $B' \subseteq B(t_0)$ and $B' \neq B_0$, then $B'(t)$ is empty. Note that if $B_0(t)$ and $B'(t)$ are both non-empty, then $n_0 > n$. (The blocks defined by P_0 are subsets of the blocks of the original computation. Therefore, we have at least n such blocks at time t . But since at least two blocks of P_0 give $B(t)$, n_0 must be strictly greater than n .) This means that $n_0 > p - k$. But B_0 traversed a copy of $\mathcal{GS}(x_1, x_2, \dots, x_k)$ with more non-empty blocks allowed by the induction hypothesis of the Main Lemma. We therefore have a contradiction and so $B(t) = B_0(t)$.

■

4.5 Proof of Theorem 1

From the Main Lemma, we know that, for any NN-JAG with p pebbles and Q states, we can find d and x_1, \dots, x_p such that if we run the NN-JAG on $\mathcal{GS}(x_1, \dots, x_p)$, no block traverses $\mathcal{GS}(x_1, \dots, x_p)$ as long as 1 block remains empty. This means that we can find a graph with $(36p^2 \log(pQ) + 1)^p$ nodes that is not totally explored by the NN-JAG.

This means that if the NN-JAG J correctly solves st -connectivity on all n node graphs, then n must be less than $(36p^2 \log(pQ) + 1)^p$. Therefore

$$\log n \leq p \log(36p^2 \log(pQ) + 1)$$

and

$$p \geq \frac{\log n}{\log 36 + 2 \log p + \log \log(pQ)}$$

Now, if $p > \log n$ or $\log Q > \log^2 n$ we already have the theorem, so we can assume that $p \leq \log n$ and $Q \leq \log^2 n$ and so

$$p \geq \frac{\log n}{\log 36 + 2 \log \log n + \log \log(\log^3 n)}.$$

Thus $p \in \Omega(\frac{\log n}{\log \log n})$ and the theorem follows.

Chapter 5

Time-space tradeoffs

5.1 Introduction

This chapter presents two results that prove time-space tradeoffs for solving graph connectivity problems. Given a JAG algorithm that uses space $S(n)$ and runs in time bounded by $T(n)$, we can compute the time-space product for the algorithm $T(n)S(n)$. What we show in this chapter is that, for some problems, we can bound TS from below. In fact both results prove lower bounds on the time required to solve the problem.

5.2 A tradeoff for JAGs

In this section, we will prove the following theorem due to Barnes and Edmonds [1]:

Theorem 4. JAGs with p pebbles require time $\Omega\left(\frac{n^2}{p \log^2(n/p)}\right)$ to solve directed st -connectivity.

From this we can derive the lower bound $TS \in \Omega\left(\frac{n^2 \log n}{p \log^2(n/p)}\right)$ since the space used by the JAG, $p \log n + \log q$ is $\Omega(\log n)$.

5.2.1 The graphs

The graphs that are used to prove the lower bound help us deal with interaction in a very interesting way. As seen in Chapter 4, a JAG must expend a lot of effort to fully explore a binary tree. In a tree there is only one path to any node and so chance collisions are more unlikely. The problem with trees is that they contain many nodes (usually exponential in their depth). Barnes and Edmonds solved this problem by coming up with a graph that has relatively few nodes, but to a JAG, is indistinguishable from a tree.

Layered Graphs

Each member of the set of layered graphs on $n = dw + 1$ nodes, $\mathcal{L}(d, w)$, has dw nodes laid out in a d by w grid with w nodes in each row (layer) and d layers. Nodes in the grid will be referenced by their grid position (row, column). There is a special vertex t (the n th node) that may or may not be attached to some node in the last row. The edges of a layered graph are of two types:

- Cross-edges – These form a path from node $(1, 1)$ to node $(1, w)$ along the first row of the graph.
- Down-edges – Every node of grid level i has associated with it two edges to two nodes of level $i + 1$. To form the different layered graphs of $\mathcal{L}(d, w)$ we will vary how these edges are chosen.

Figure 5-1 shows a graph from $\mathcal{L}(3, 4)$.

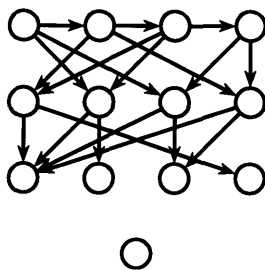


Figure 5-1: A layered graph

When solving st -connectivity on layered graphs, the start vertex s is node $(1,1)$ and the goal vertex is t .

Generalizing Layered Graphs – k -tree Graphs

As previously mentioned, layered graphs will “look” like trees to the JAG. We will build up a good layered graph in stages. The following family of graphs, called k -trees will be useful.

A k -tree with width w and height h looks much like a layered graph with the same width and height. The first k layers are the same as for a layered graph. The differences are on levels k through h . In a k -tree, each node on level k is the root of a different complete binary tree of height $h - k + 1$. For example, a 1-tree consists of w binary trees of height h with the roots connected by a line. See Figure 5-2 for a diagram of a k -tree.

5.2.2 The proof

The proof shows that if a JAG claims to visit all nodes on the last level of all n node layered graphs with width w and height h , then it must use more than $\min(w2^h, \frac{(w-1)^2}{2(p-1)})$ steps to do so. We will build up a difficult layered graph by running the JAG on k -tree graphs that have more than n nodes. This may seem a little unfair, but we maintain the invariant that on all of the graphs that we construct, the sequence of state-coincidence partition pairs stays the same for the first $w2^h$ steps. If we manage to keep the coincidence partition the same then the JAG doesn't know that the k -tree it's running on has more nodes that it can “handle”.

The theorem is proved using the following lemma:

Lemma 6. If the JAG J runs for fewer than $\min(w2^h, \frac{(w-1)^2}{2(p-1)})$ steps, then there is a leaf of some k -tree ($1 \leq k \leq h$) that isn't visited.

Proof:

We prove this by induction on k . For the base case, k is 1. Recall that a 1-tree consists of w trees of height h . The total number of leaves in this graph is $w2^h$.

The JAG can't possibly visit all of them in fewer than $w2^h$ steps. So there is a distinguished vertex v^* that J does not visit.

To prove the induction step, we are given a $k - 1$ tree and the vertex v^* that J does not visit. To construct a k -tree, we observe that "level" k of a $k - 1$ tree consists of $2w$ nodes since this level is the first level of each of the binary trees. We also note that these nodes are the roots of $2w$ binary trees (see Figure 5-2). If we can delete half of these trees, by connecting the nodes of level $k - 1$ to only w of them (so that each node is connected to 2 trees), we will have a k -tree. The challenge is to find such a tree with the required property.

Given the $k - 1$ tree, we can identify the $2w$ binary trees that are rooted at level k , (T_1, \dots, T_{2w}) . What we want to do is to break up the trees into w disjoint sets where all trees in each set are "equivalent". Trees T_i and T_j are "equivalent" if, at no time during the computation, the JAG has a pebble on T_i and a pebble on T_j . We now show how to find such a partition that is small.

We can construct the undirected graph H where the nodes represent the trees T_1, \dots, T_{2w} and an edge between node i and j means that there is a time in the JAG computation where there was a pebble on T_i and a pebble on T_j . If we can colour this graph with c colours, we can find a partition with c sets (just put all of the trees with the same colour in the same set). By the construction of the graph, we are guaranteed that all trees in each set are "equivalent".

To bound the chromatic number of H , we first observe that H has at most $\frac{(w-1)^2}{2}$ edges. To see this note that we add at most $p - 1$ edges to H every time a move is made. Each time we move a pebble, there are $p - 1$ other pebbles that need to be checked. This means that there are only $p - 1$ trees that need to be checked to see whether we add edges to H . Since we run for time bounded by $\frac{(w-1)^2}{2(p-1)}$, and we add at most $p - 1$ edges at each time step, we have at most $\frac{(w-1)^2}{2}$ edges in H .

We next use the fact that a graph with E edges can be coloured with $\sqrt{2E}$ colours. This implies that we can colour H with $w - 1$ colours. Therefore, we can partition the trees into $w - 1$ sets. We know that there is some tree T_u and some leaf v^* of T_u that isn't visited by the JAG during the computation on the $k - 1$ tree. We want

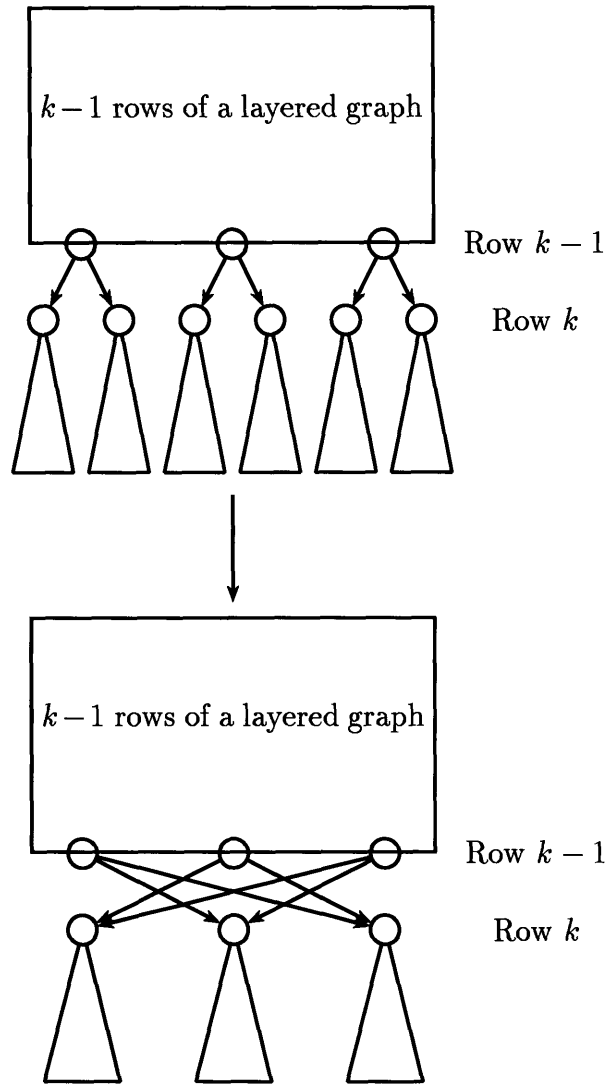


Figure 5-2: The transformation we hope to achieve

to keep this invariant, so we put T_u in a separate class. We now create the k tree by picking one member of each class and replacing links to any member of that class with links to that distinguished member.

We now note that the sequence of state-coincidence partition pairs is the same for this k -tree as for the $k - 1$ -tree that we started out with and that v^* isn't visited. The only way the coincidence partitions could differ is if two pebbles meet in some tree in G_k but don't meet in G_{k-1} . However, from the way we chose the partition this can't happen. Collisions in G_k are the same as those in G_{k-1} .

■

Using this lemma we can find a layered graph with n nodes and a leaf v^* that isn't visited by the JAG. Therefore, the answer given by the JAG (“ s and t are/aren't connected”) doesn't change if we add or remove an edge from v^* to t . Therefore the JAG makes a mistake. So JAGs need more than $\min(w2^h, \frac{(w-1)^2}{2(p-1)})$ to solve directed st -connectivity. Setting $w = \frac{n}{\log(n/p)}$ gives a time bound of $\Omega(\frac{n^2}{p \log^2(n/p)})$.

5.3 A tradeoff for modified NJAGs

Our final result proves a lower bound of $\Omega(n^2/p)$ on the time it takes for a modified nondeterministic JAG to solve st -nonconnectivity on a class of undirected 3-regular graphs. The NJAGs that are discussed in this section have the following additional features:

- $p - 1$ unmovable pebbles and 1 active pebble. The NJAG is given p pebbles, but can only move 1 of them, the active pebble. The other $p - 1$ pebbles are fixed—once placed they cannot be moved during the computation
- Strong Jumping. The NJAG can jump the active pebble to any node in the graph (not just nodes with pebbles on them).

This result was proved by Beame et al. [2] and uses following idea: There will be a path from s to t in all but 1 of the graphs. The fact that the NJAG must answer “yes” on one graph forces it to have a long computation on that graph.

5.3.1 The graphs

The basic graph that we work with here is a “squirrel cage” graph.

Definition. A squirrel cage graph with k nodes consists of two $k/2$ cycles with edges joining node i of the first cycle with node i of the second cycle for $1 \leq i \leq k/2$. Intuitively, we can view the graph as forming a “cage” (see Figure 5-3)

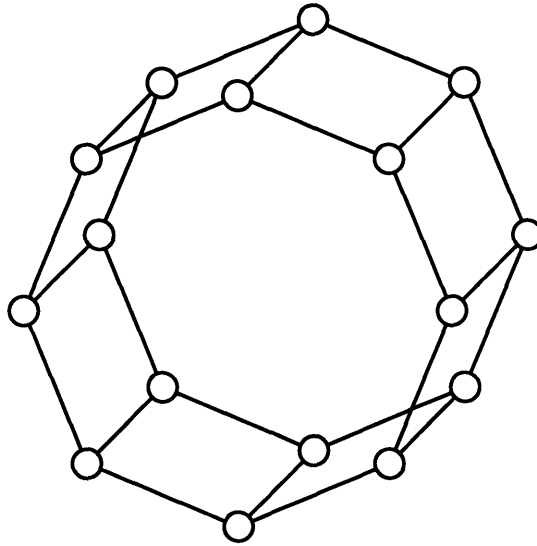


Figure 5-3: A squirrel cage graph with 16 nodes

To construct our n node graphs, we first take 2 squirrel cage graphs with $n/2$ nodes, S^0 and S^1 . We then fix $r = n/4 - 1$ of the inter-cycle edges in S^0 (and note the corresponding edges in S^1). We call these edges switchable edges. Given a binary string x of length r we will construct the graph G_x in the following way:

We order the switchable edge indices from 1 to r . We then replace the switchable edges (u_i^0, v_i^0) in S^0 , and (u_i^1, v_i^1) in S^1 , with $(u_i^0, v_i^0 \text{ XOR } x_i)$ and $(u_i^1, v_i^1 \text{ XOR } x_i)$. We basically “switch” some of the edges in the squirrel cage graph. Note that if there is at least 1 bit that is set to 1 in x , then the graph G_x is connected and so the only graph that isn’t connected is G_{0^r} . The start node s is any node in S^0 and t is any node in S^1 . Figure 5-4 shows such a graph.

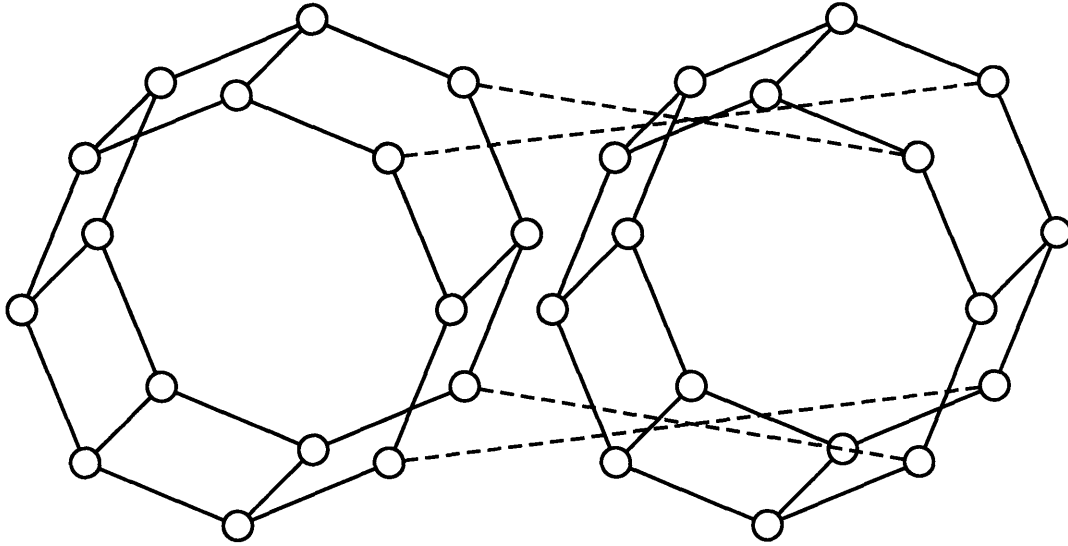


Figure 5-4: A sample input graph (the dashed edges have been switched)

5.3.2 The proof

As with the proof described in Chapter 2, we will identify important features of the computation of the NJAG on G_{0^r} . As before, we will view the computation history of the NJAG as a sequence of state-partition pairs. Since a NJAG learns about the graph by colliding with other pebbles (the unmovable ones), we will mark the nodes that have unmovable pebbles on them. We will actually go one step further and mark node i in both S^0 and S^1 if node i in any of S^0 or S^1 contains an unmovable pebble.

The important features of the computation history of J on G_{0^r} (denoted by \mathcal{H}) are walks that are between marked nodes, for J learns that the two marked nodes are in the same connected component. Formally we will be interested in sequences of moves that:

- Begin with the first move from a marked vertex or a vertex that was just jumped to (in the previous move).
- End when the next move lands us at a marked vertex, is a jump, or the computation history ends.

Note that all IDs of \mathcal{H} don't necessarily appear in some sequence. If we have a

sequence of jumps, then they don't appear in a sequence.

The above procedure allows us to break up \mathcal{H} into l sequences $\mathcal{H}_1, \dots, \mathcal{H}_l$.

Consider the moves of \mathcal{H}_i (for any i). They are all walks and we can attempt to follow \mathcal{H}_i on graph G_x . Since no marked vertices are reached, and each node along the way has the same degree, \mathcal{H}_i can indeed be followed on graph G_x . If we cross from S^0 to S^1 and from S^1 to S^0 an even number of times, we end up on the same S^b ($b = 0, 1$) as we started out with. In fact, we will end up on the same node in G_x as in G_{0^r} .

Lemma. Given x in $\{0, 1\}^r$, if we cross an even number of “switched” edges in G_x in all \mathcal{H}_i , then the computation history \mathcal{H} is also a computation history for J on G_x .

Proof: The lemma is proved by proving the following inductive statement: For $1 \leq i \leq l$ the portion of \mathcal{H} from the beginning till the end of \mathcal{H}_i is a computation history for J on G_x if number of switched edge crossings in \mathcal{H}_k is even for $1 \leq k < i$.

The base case of this proof is when $i = 0$. There is nothing to prove here. For the induction step assume that everything agrees up to the end of \mathcal{H}_i . We will now show that the history agrees up to the end of \mathcal{H}_{i+1} . Note that the positions of the pebbles at the end of \mathcal{H}_i are the same in G_x and G_{0^r} . So get from the end of \mathcal{H}_i to the beginning of \mathcal{H}_{i+1} we do some jumps. We know that all of the unmovable pebbles are in the same positions so we start \mathcal{H}_{i+1} in the same configuration. Since all of the moves of \mathcal{H}_{i+1} are walks and we cross an even number of switched edges, the active pebble will end up on the same node in G_{0^r} and G_x . Thus the computation histories agree up to the end of \mathcal{H}_{i+1} .

Note that after the last sequence \mathcal{H}_l only jumps or a walk that doesn't end in a marked vertex is possible and in this case the partitions agree since the degree of each vertex is the same. ■

Since we accept G_{0^r} and reject G_x (for $x \neq 0^r$) we must ensure that we never satisfy the hypothesis of the above lemma. There is an interesting linear algebra formulation of this requirement. If i_1, \dots, i_t are the switchable edges that occur in

\mathcal{H}_i we know that:

$$x_{i_1} + x_{i_2} + \dots + x_{i_l} \equiv 0 \pmod{2}$$

Since the bits of x are 0 or 1, this gives us an equation over $GF(2)$. This equation must hold for all of the sequences and so we have a system of l linear equations in r unknowns over $GF(2)$. Note also that this system is homogeneous (the right side is zero).

We now show that the walks that comprise the sequence \mathcal{H}_i are long. To do this, some more definitions are needed:

Definition. Given a switchable edge e , $\text{dist}(e)$ is the “distance” from e to the closest node that is marked. The distance from an edge to a node is defined here to be the length of the shortest path that contains both the node and the edge.

Definition. The set S_d contains all switchable edges e with $\text{dist}(e) \geq d$. $\text{max}d$ is the maximum distance of any switchable edge. r_d denotes the number of switchable edges with distance exactly d . Note that $r_d = |S_d| - |S_{d+1}|$

Consider the number of walks of length at least d . We claim that it is bounded below by $|S_d|$. To see this note that if fewer than $|S_d|$ walks have length at least d , then the variables (bits of x) that correspond to the edges in S_d are represented in fewer than $|S_d|$ of the equations of our system. Therefore, setting the other $r - |S_d|$ bits to be zero, we have a system of fewer than $|S_d|$ equations in $|S_d|$ unknowns. Systems with fewer equations than unknowns always have non-trivial solutions, and so we violate our requirement of no non-zero solutions to the system.

Therefore at least $|S_d|$ of the sequences are walks of length at least d . From the above paragraph we can infer that r_d lower bounds the number of walks of length d . Thus the total number of walks that the NJAG makes is at least:

$$\sum_{d=1}^{\text{max}d} dr_d$$

Since r_d is the number of switchable edges with distance d , the sum equals

$$\sum_{e \text{ a switchable edge}} \text{dist}(e)$$

We now want to lower bound the above sum. It turns out that the minimum value of the sum occurs when all of the marked nodes are equally spaced in each cycle. In this case the sum is $\Omega(rn/p) = \Omega(n^2/p)$. This proves the theorem.

It is interesting to note that in this proof we don't construct the graph to "fool" the NJAG in stages as in previous results. The graph is fixed and we prove the lower bound no matter what NJAG we are presented with.

Chapter 6

Conclusion

Many other results concerning JAGs and their variants have been proved. This chapter attempts to summarize some of them and present some interesting open problems.

6.1 Other Results

6.1.1 Probabilistic JAGs

As with Turing Machines, probabilistic analogues of JAGs can be defined. Berman and Simon [3] defined one such model where there are 2 functions (f_1 and f_2) that determine the next move. f_1 takes a state and the coincidence partition and outputs the probability distribution on the next state. f_2 takes the next state and determines the move made (walk or jump). On this model they proved a space lower bound of $\Omega(\log^2 n / \log \log n)$ for solving directed st -connectivity in time $O(n^{\log n^c})$ ($c > 0$) with one-sided error less than $1/2$.

Poon [6] also extended his NN-JAG model with probabilism in a slightly different way. An instantaneous description of a probabilistic NN-JAG contains the current time as well as the state and pebble mapping (t, q, P) . Based on this and the random string given to the NN-JAG (the entire string, not just single bit) the next move and state is computed. A space lower bound of $\log^2 n / (10 + c) \log \log n$ is shown for this NN-JAG to solve directed st -connectivity in time $2^{\log^c n}$ with one-sided error less than

1/2.

6.1.2 Tradeoffs

The “state of the art” in time-space tradeoffs for directed st -connectivity appears in [1]. Along with the result presented in Chapter 5, they show a tradeoff of $S^{1/3}T = \Omega(m^{2/3}n^{2/3})$ for NN-JAGs on n -node, m -edge graphs.

6.1.3 Directed st -nonconnectivity

Poon [6] showed that nondeterministic NO-JAGs (and hence nondeterministic NN-JAGs) can perform “inductive counting” using logarithmic space and so can solve st -nonconnectivity using logarithmic space. This can also be interpreted as further evidence that JAGs are quite powerful models of computation.

6.1.4 Undirected Graphs

In his PhD thesis Edmonds [5], viewing a probabilistic JAG as a distribution on JAG algorithms proved that if we have $O(\log n / \log \log n)$ pebbles and $2^{\log^{O(1)} n}$ states then the expected time for a such a JAG to solve undirected st -connectivity is $n2^{\Omega(\log n / \log \log n)}$. Beame et. al. [2] prove many other results dealing with the complexity of traversing undirected graphs.

6.2 The future

While improving on the bounds described above is interesting and worthwhile, other challenges exist. Chapter 2 introduced the problem of solving st -nonconnectivity on NJAGs. Matching upper and lower bounds (for space) were shown for the 1-pebble case. However, we know no good bounds when we don’t restrict the number of pebbles.

In the end of their paper Barnes and Edmonds [1] suggest that we consider what additional power we can add to a JAG while still being able to prove superlogarithmic

space bounds. They note that Poon did just this by inventing NO-JAGs and NN-JAGs. This seems to be a very plausible way to approach the eventual goal of proving lower bounds on Turing Machines.

Bibliography

- [1] Greg Barnes and Jeff A. Edmonds. Time-space lower bounds for directed s - t connectivity on JAG models. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*. IEEE, November 1993.
- [2] Paul Beame, Allan Borodin, Prabhakar Raghavan, Walter L. Ruzzo, and Martin Tompa. Time-space tradeoffs for undirected graph traversal. Technical Report 93-02-01, Department of Computer Science and Engineering, University of Washington, February 1993.
- [3] Piotr Berman and Janos Simon. Lower bounds on graph threading by probabilistic machines. In *Proceedings of the 24th Annual Symposium on Foundations of Computer Science*. IEEE, November 1983.
- [4] Stephen A. Cook and Charles W. Rackoff. Space lower bounds for maze threadability on restricted machines. *SIAM Journal on Computing*, 9(3):636–652, August 1980.
- [5] Jeff A. Edmonds. *Time-Space Lower Bounds for Undirected and Directed ST-Connectivity on JAG Models*. PhD thesis, University of Toronto, 1993.
- [6] C. K. Poon. Space bounds for graph connectivity problems on node-named JAGs and node-ordered JAGs. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*. IEEE, November 1993.
- [7] Avi Wigderson. The complexity of graph connectivity. In *Proceedings of the 17th International Symposium on Mathematical Foundations of Computer Science*, August 1992.