Approximating Longest Path

Björklund, Andreas

2003

Approximating Longest Path

Andreas Björklund

Supervisor: Thore Husfeldt

Abstract

We investigate the computational hardness of approximating the longest path and the longest cycle in undirected and directed graphs on n vertices. We show that

- in any expander graph, we can find $\Omega(n)$ long paths in polynomial time.
- there is an algorithm that finds a path of length $\Omega(\log^2 L/\log\log L)$ in any undirected graph having a path of length L, in polynomial time.
- there is an algorithm that finds a path of length $\Omega(\log^2 n/\log\log n)$ in any Hamiltonian directed graph of constant bounded outdegree, in polynomial time.
- there cannot be an algorithm finding paths of length $\Omega(n^\epsilon)$ for any constant $\epsilon > 0$ in a Hamiltonian directed graph of bounded outdegree in polynomial time, unless $P = NP$.
- there cannot be an algorithm finding paths of length $\Omega(\log^{2+\epsilon} n)$, or cycles of length $\Omega(\log^{1+\epsilon} n)$ for any constant $\epsilon > 0$ in a Hamiltonian directed graph of constant bounded outdegree in polynomial time, unless 3-Sat can be solved in subexponential time.

Preface

The research presented in this work is my own, and was done under the supervision of Thore Husfeldt. The results were announced previously in

- A. Björklund and T. Husfeldt. *Finding a Path of Superlogarithmic Length.* SIAM Journal on Computing. Vol. 32 (2003), No. 6, pp. 1395-1402. A preliminary version was presented in the proceedings of the 29th International Colloquium on Automata, Languages and Programming (2002), pp. 985-992.
- A. Björklund, T. Husfeldt, and S. Khanna. *Approximating Longest Directed Path.* Electronic Colloquium on Computational Complexity, Report TR03-032.

My own contribution to the results presented here is substantial; especially, the basic ideas for both the lower bound in the latter of the articles, and the algorithms in both the articles above are mine. The $n^{1-\epsilon}$-hardness of longest directed path was first shown in [16] and independently in [3]. The reduction in the latter is used in the joint paper above.

Acknowledgements

I thank Rolf Karlsson for introducing me properly to the theory of computational complexity. He and Andrzej Lingas encouraged me to do research in theoretical computer science, and always showed confidence in my abilities. So did many others at the CS department, including Anna, Björn, Jesper, Joachim, and Mikael. Andrzej also pointed me to the problem of approximating the longest path, for which I am most grateful.

A special tribute goes to to Bengt J. Nilsson who "got me back" to the department, where my work with Thore Husfeldt was initiated. I see in Thore a skilled researcher in theoretical computer science as well as a friend, which makes him the perfect supervisor. Apart from that, he has one admirable quality which I certainly lack and which has proven absolutely crucial for this work to take place: he gets things done.

I also thank all of my friends for encouraging me to carry on with my research activities. In particular I am grateful to Mats Petter Pettersson and my friends at work for always lending their ears and minds to small discussions.

Finally, I want to express my gratitude to the people dearest to me, my father, mother, sister, and my girlfriend Maria, for showing they are proud of me and their respect for my interest in math and computers, while constantly reminding me there is much more to life.

Chapter 1

The Longest Path Problem

1.1 Introduction

The central quest in the field of computational complexity is to establish how much resources is required to solve particular finite mathematical problems. For instance, the question may concern how much time is needed for a conventional computer to solve instances of a problem of given size. Note that it makes little sense to ask for the time needed to solve a particular instance of a problem, since the answer may be tabulated in the program and thereby the instance is "solved" immediately, regardless of the actual underlying problem. By asking for the behaviour of a program that solves any instance of a given problem, the situation changes. Unfortunately, it appears that proving some particular problems to be computationally hard is either impossible, or requires radically new methods, since every reasonable attempt made by mathematicians the last 40 years or so (complexity theory is a young science), have turned out to be only modestly successful. However, some progress have been made classifying problems as equally hard, if we allow ourselves to adopt a loose sense of the word "equal". We say a problem A reduces to another problem B, if it is possible to formulate any instance of A as an instance of problem B. If we can show the reduction to be efficient (i.e. much easier than actually solving problem A), then B is at least as hard as problem A. It is natural to put problems into classes. The most famous complexity classes are P and NP, where P is the class of problems solvable in time polynomial in the length of the input, and NP is the class of problems where a claimed solution may be verified in time polynomial in the length of the input. For all we know, these two classes may well be the very same class, although we have not seen much to support that hypothesis. In particular, for the subclass of NP known as the NP-complete problems, containing the problems at least as hard as *every* other problem in NP, we know of no subexponential time algorithms. On the other hand, if we could find a truly efficient algorithm for one of these problems, we would also know how to solve all of them efficiently.

Finding the longest non-crossing path in a graph is such a problem. Formally, given an unweighted graph or digraph $G = (V, A)$ with $n = |V|$, the *Longest Path* problem is to find the longest sequence of distinct vertices $v_1 \cdots v_k$ such that $v_i v_{i+1} \in A$. Closely related is the *Longest Cycle* problem, where we in

addition require that $v_k v_1 \in A$. A path or cycle visiting every vertex of the graph once is called a *Hamiltonian* path or cycle. A graph that contains a Hamiltonian cycle is called Hamiltonian. The concept owe its name to W. R. Hamilton, who in 1856 [11] showed that deductions in certain families of systems of non-commutative roots of unity had a graph problem interpretation: walk along the edges of a regular dodecahedron in such a way that every corner is passed once and only once, and you eventually end up where you started. Although finding a Hamiltonian path in that particular graph is fairly easy, one soon realized that determining if an arbitrary graph was Hamiltonian or not, seemed to require testing almost all the paths in the worst case, to see if anyone of them covered all vertices.

Interestingly, starting some twenty-five years ago, mathematicians in the field of random graphs have found increasingly stronger evidence that the Hamiltonicity problem is easy *on the average*, [2] [10] [6]. To exemplify, the result of [6] shows an algorithm which is likely to determine in polynomial time whether the random graph G_p where each edge exists with independent probability $p = d/n$ for some (large enough) constant d , is Hamiltonian or not. However, all these methods require exponential time in the worst case. Indeed, the Hamiltonicity problem is NP-complete [15], and thus it is not expected to admit subexponential time algorithms.

A natural extension is to explore how long paths a polynomial time algorithm can guarantee to find, relative to the longest path in the graph, i.e. approximating the longest path. The measure used as an approximation guarantee is the maximum value of the ratio between the optimum length and the length of the approximate solution found, taken over all instances of the same size n. The *Longest Path* problem in undirected graphs is notorious for the difficulty of understanding its approximation hardness [7]. However, we know that it cannot be approximated within any constant factor of the optimum in polynomial time unless $P = NP$, or within $2^{O(\log^{1-\epsilon} n)}$ for any constant $\epsilon > 0$ unless $NP \in DTIME(2^{\log^{1/\epsilon} n})$ [14]. For a long time, the best algorithms finding long paths in polynomial time, only guaranteed to find paths of length $\Omega(\log L / \log\log L)$, where L is the length of the longest path [18] [4]. A significant improvement was presented in [1], where it was shown how to find a path or cycle of logarithmic length in polynomial time, whenever there exists one. The best polynomial time approximation algorithm for longest path in undirected graphs known to date, to the best of my knowledge, is the one presented in §2.2, which finds paths of length at least $\Omega(\log^2 L / \log\log L)$. This result though, may very well be far from best possible. Recently, Feder, Motwani and Subi [7], showed that in graphs where all vertices are adjacent to at most three others, it is possible to approximate *Longest Cycle* within $O(n^{1-(\log_3 2)/2})$, in polynomial time.

When we restrict ourselves to directed graphs, it is of course only getting worse on the algorithmic side. The algorithms in [18] and [1] work also for directed graphs, whereas our algorithm presented in §2.2 cannot be used. However, in the special case of bounded outdegree Hamiltonian directed graphs we can do just as good as shown in §2.3. The hardness results presented in §3, provide evidence why we should not expect to find a much better algorithm than the one in §2.3. §3 also shows that the *Longest Cycle* problem appears to be even harder. The best algorithm for that problem was found recently by

Gabow and Nie [9], almost matching our lower bound.

1.2 Preliminaries

A graph or directed graph is a tuple $G = (V, E)$ or $G = (V, A)$ respectively, where V is the vertex set sometimes identified with $\{1, 2, \ldots, n\}$. The edge set E and arc set A, are subsets of $V \times V$. We write uv for the edge or arc (u, v). For a subset $W \subseteq V$ of the vertices of a graph G, we denote by $G[W]$ the graph induced by W, i.e. the graph consisting of the vertices W and all edges or arcs between the vertices W.

The *length* of a path and a cycle is its number of edges. The length of a cycle C is denoted $l(C)$. A k-cycle is a cycle of length k, a k^+-cycle is a cycle of length k or larger. A k-path and k^+-path is defined similarly. For vertices x and y, an xy-path is a simple (non-crossing) path from x to y, and if P is a path containing u and v we write $P[u, v]$ for the subpath from u to v. We let $L_G(v)$ denote the length of the longest path from a vertex v in the graph G. The *path length* of G is $L = \max_{v \in V} L_G(v)$.

Chapter 2

Algorithms Finding Long Paths

A natural step when investigating the computational hardness of a problem, is to try to solve it as efficiently you can, to get an upper bound on its complexity. In this chapter we present three algorithms, one finding linear length paths in sparse expander graphs, one finding superlogarithmic long paths in general undirected graphs, and one finding superlogarithmic long paths in directed Hamiltonian graphs of bounded outdegree.

2.1 Finding Long Paths in Most Sparse Graphs

The fact that there has been little success in presenting algorithms to approximate the longest path in general, does not imply that the problem is hard on the average, as mentioned in chapter 1. On the contrary, whenever the graph is tightly connected, i.e. when there is a sufficient amount of arcs or edges leading out from every subset of the vertices, it is easy to find a long path.

A digraph $G = (V, A)$ on n vertices is a c-expander if $|\delta U| \geq c(1 - \frac{|U|}{n})|U|$ for every subset $U \subset V$ where $\delta U = \{ v \notin U \mid \exists u \in U : uv \in A \}$.

A standard probabilistic argument (see e.g. [17] for a similar proof) shows that with high probability a random digraphs with outdegree k ($k > 2$), are c_k-expanders for some constant c_k, for large enough $n > n_k$. In other words, the result holds for most bounded outdegree graphs.

We propose the following algorithm for finding a long path $p_0 \cdots p_l$ in a sparse expander.

1. Pick an arbitrary start vertex p_0, and set $i = 0$.
2. Let $G_i = (V_i, A_i)$ be the subgraph reachable from p_i in $G[V \setminus (\bigcup_{j=0}^{i-1} p_j)]$.
3. If G_i consists only of p_i, exit.
4. For each neighbour v of p_i in G_i, evaluate the size of the subgraph reachable from v in $G_i[V_i \setminus p_i]$.
5. Choose the neighbour who has the largest reachable subgraph as p_{i+1}.
6. Set $i = i + 1$ and goto 2.

Theorem 1 *The algorithm finds a path of length* $\frac{c}{2(k+1)}n$ *in every* c*-expander digraph* $G = (V, A)$ *with maximum outdegree* k*.*

Proof. Consider step i. Enumerate the neighbours of p_i in G_i as $r_1 \cdots r_{k'}$. Let $V_i[r_j]$ be the vertices reachable from r_j in $G_i[V_i - \{p_i\}]$. Now observe that the $V_i[r_j]$ either are very small or really large for small i, since the set of vertices outside $V_i[r_j]$ in G which are directly connected by an arc from a vertex in $V_i[r_j]$ must lie on the prefix path $p_0 \cdots p_i$ by definition, and there must be a lot of them because of the expander criterion. Specifically, when i is small, there must be a j for which $V_i[r_j]$ is large, since $k' \leq k$ and $\bigcup V_i[r_j] = V_i - \{p_i\}$. Observe that V_{i+1} is the largest $V_i[r_j]$, to obtain

$$|V_{i+1}| \geq n - \frac{2(i+1)}{c}$$

whenever at least one $V_i[r_j]$ is too large to be a small subgraph, i.e. as long as

$$\frac{c(|V_i| - 1)}{2k} \geq i + 1,$$

where we for the sake of simplicity have used the expansion factor $c/2$ which holds for all set sizes. Observing that $V_0 = n$, we may solve for the smallest i, when the inequality above fails to hold. This will not happen unless $i \geq \frac{c}{2(k+1)}n$, as promised. ∎

2.2 Finding a Long Path in an Undirected Graph

We do not know of any reason why it would be impossible to find paths of length say $\sqrt{L}$ where L denotes the length of the longest path in a general undirected graph in polynomial time. Especially, the lower bound of [14] does not impose any unexpected consequences if such an algorithm exists. Still, until recently, we only knew how to find a path of logarithmic length [1] in polynomial time. Vishwanathan [22] obtained a slight improvement when he showed how to find superlogarithmic long paths whenever the graph is Hamiltonian. We describe in this section how to get rid of the Hamiltonicity assumption, by using a completely different algorithm.

Theorem 2 *If a graph contains a simple path of length* L *then we can find a simple path of length*

$$\Omega\left(\frac{\log^2 L}{\log \log L}\right)$$

in polynomial time.

Our algorithmic idea is to look for a connected string of long enough cycles. By taking the longest way round every cycle we obtain our long path. For the purpose of finding these cycles, we need the following recent result [9],

Theorem 3 (Gabow and Nie) *Given a graph, one of its vertices* s*, and an integer* k*, one can find a* k^+*-path passing through* s *(if it exists) in time* $2^{O(k)}n \log n$*.*

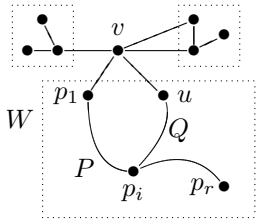

Figure 2.1: Statement 1 of Lemma 2. The path $P = vp_1 \cdots p_r$ continues in the component W. We assume that v does not lie on a large cycle. This means that an arbitrary path Q from v's neighbour u must intersect P 'early,' i.e., $QP[p_i, p_r]$ is long.

We also need the following easy lemma.

Lemma 1 *If a connected graph contains a path of length r then every vertex is an endpoint of a path of length at least $\frac{1}{2}r$.*

Proof. Given vertices $u, v \in V$ let $d(u, v)$ denote the length of the shortest path between u and v.

Let $P = p_0 \cdots p_r$ be a path and let v be a vertex. Find i minimising $d(p_i, v)$. By minimality there is a path Q from v to p_i that contains no other vertices from P. Now either $QP[p_i, p_r]$ or $QP[p_i, p_0]$ has length at least $\frac{1}{2}r$. ∎

The next lemma is central to our construction: Assume that a vertex v originates a long path P and v lies on a cycle C; then the removal of C decomposes G into connected components, one of which must contain a large part of P. See Figures 2.1 and 2.2.

Lemma 2 *Assume that a connected graph G contains a simple path P of length $L_G(v) > 1$ originating in vertex v. There exists a connected component $G[W]$ of $G[V - v]$ such that the following holds.*

1. *If $G[W + v]$ contains no k^+-cycle through v then every neighbour $u \in W$ of v is the endpoint of a path of length*

$$L_{G[W]}(u) \geq L_G(v) - k.$$

2. *If C is a cycle in $G[W + v]$ through v of length $l(C) < L_{G[W+v]}(v)$ then there exists a connected component H of $G[W - C]$ that contains a neighbour u of $C - v$ in $G[W + v]$. Moreover, every such neighbour u is the endpoint of a path in H of length*

$$L_H(u) \geq \frac{L_G(v)}{2l(C)} - 1.$$

Proof. Let $r = L_G(v)$ and $P = p_0 \cdots p_r$, where $p_0 = v$. Note that $P[p_1, p_r]$ lies entirely in one of the components $G[W]$ of $G[V - v]$.

First consider statement 1; see Fig. 2.1. Let $u \in W$ be a neighbour of v. Since $G[W]$ is connected, there exists a path Q from u to some vertex of P. Consider such a path. The first vertex p_i of P encountered on Q must have

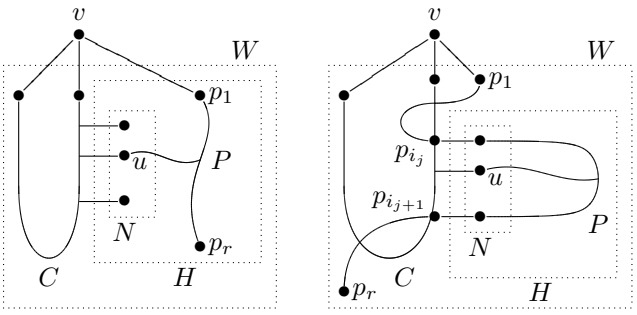

Figure 2.2: Statement 2 of Lemma 2. Here we assume that v does lie on a large cycle C. In case 1 (left) the path $P = vp_1 \cdots p_r$ does not intersect C after it leaves v. Thus $P[p_1, p_r]$ lies entirely in a component H of $W - C$. Any neighbour $u \in N$ of C in this component must be the head of a long path using at least half of $P[p_1, p_r]$. In case 2 (right) the path P intersects C in several places. Consider the largest section of P that lies entirely in a component H of $W - C$, here shown as a 'loop' starting after p_{i_j} and ending before $p_{i_{j+1}}$. Any neighbour $u \in N$ of C in this component must be the head of a long path using at least half of the 'loop.'

$i < k$ since otherwise the three paths vu, $Q[u, p_i]$ and $P[p_0, p_i]$ form a k^+-cycle. Thus the path $Q[u, p_i]P[p_i, p_r]$ has length at least $r - k + 1 > r - k$.

We proceed to statement 2; see Fig. 2.2. Consider any cycle C in $G[W + v]$ through v.

Case 1. First assume that $P \cap C = v$, so that one component H of $G[W - C]$ contains all of P except v. Let N be the set of neighbours of $C - v$ in H. First note that N is nonempty, since $G[W]$ is connected. Furthermore, the path length of H is at least $r - 1$, so Lemma 1 gives $L_H(u) \geq (r - 1)/2$ for every $u \in N$.

Case 2. Assume instead that $|P \cap C| = s > 1$. Enumerate the vertices on P from 0 to r and let $i_1, \ldots, i_s$ denote the indices of vertices in $P \cap C$, in particular $i_1 = 0$. Let $i_{s+1} = r$. An averaging argument shows that there exists j such that $i_{j+1} - i_j \geq r/s$. Consequently there exists a connected component H of $G[W - C]$ containing a simple path of length $r/s - 2$. At least one of the i_jth or i_{j+1}th vertices of P must belong to $C - v$, so the set of neighbours N of $C - v$ in H must be nonempty. As before, Lemma 1 ensures $L_H(u) \geq r/2s - 1$ for every $u \in N$, which establishes the bound after noting that $s \leq l(C)$. ∎

We first give a brief overview of the algorithm, the next two sections will provide the details.

Assume for simplicity that the input graph is connected; this is no restriction since otherwise we can iterate the algorithm over each connected component of the input graph and return the longest path found. Pick any vertex v. Lemma 1 ensures that v is the head of a path of length at least $r > L/2$. In the next sections we will pretend that we know the value

$$k = \lceil \tfrac{1}{4} \log r \rceil$$

but this is no restriction since we can (in polynomial time) run the algorithm for every value of $k = 3, \ldots, \lceil \frac{1}{4} \log n \rceil$ and return the longest path found.

Given v and k we will construct a tree $T_k(G, v)$ as detailed in §2.2.1; this tree will describe a recursive decomposition of the input graph G into paths and cycles. Finally, we find a long (weighted) path in $T_k(G, v)$. This path will describe a path in G which will have the desired length as shown in §2.2.2.

In summary, the algorithm proceeds as follows, assuming a connected input graph:

1. Pick any vertex $v \in G$.
2. For every $k = 3, \ldots, \lceil \frac{1}{4} \log n \rceil$ perform the following two steps and return the longest path found:
3. Construct the tree $T_k(G, v)$ as detailed in §2.2.1.
4. Find a longest weighted path in $T_k(G, v)$ and return the path in G described by it, as detailed in §2.2.2.

Steps 3 and 4 take polynomial time (see below), so the entire algorithm takes polynomial time.

2.2.1 Construction of the Cycle Decomposition Tree

Given a vertex v in G, our algorithm constructs a rooted node-weighted tree $T_k = T_k(G, v)$, the cycle decomposition tree. Every node of T_k is either a *singleton* or a *cycle* node: A singleton node corresponds to a single vertex $u \in G$ and is denoted $\langle u \rangle$, a cycle node corresponds to a cycle C with a specified vertex $u \in C$ and is denoted $\langle C, u \rangle$. Every singleton node has unit weight and every cycle node $\langle C, u \rangle$ has weight $\frac{1}{2} l(C)$.

The tree $T_k(G, v)$ is constructed as follows. Initially T_k contains a singleton node $\langle v \rangle$, and a call is made to the following procedure with arguments G and v.

1. [Iterate over components:] For every maximal connected component $G[W]$ of $G[V - v]$, execute step 2.
2. [Find cycle:] Search for a k^+-cycle through v in $G[W + v]$ using Theorem 3. If such a cycle C is found then execute step 3, otherwise execute step 5.
3. [Insert cycle node:] Insert the cycle node $\langle C, v \rangle$ and the tree edge $\langle v \rangle \langle C, v \rangle$. For every connected component H of $G[W - C]$ execute step 4.
4. [Recurse:] Choose an arbitrary neighbour $u \in H$ of $C - v$, and insert the singleton node $\langle u \rangle$ and the tree edge $\langle u \rangle \langle C, v \rangle$. Then, recursively execute step 1 to compute $T_k(H, u)$.
5. [Insert singleton node and recurse:] Pick an arbitrary neighbour $u \in G[W + v]$ of v, insert the node $\langle u \rangle$ and the tree edge $\langle v \rangle \langle u \rangle$, and recursively execute step 1 to compute $T_k\big(G[W], u\big)$.

Note that each recursive step constructs a tree that is connected to other trees by a single edge, so T_k is indeed a tree. Also note that the ancestor of every cycle node must be a singleton node. The root of T_k is $\langle v \rangle$.

To see that the running time of this procedure is polynomial first note that step 2 is polynomial because of Thm. 3. The number of recursive steps is linear, since every step inserts a node into T_k, which is clearly of linear size after the procedure.

2.2.2 Paths in the Cycle Decomposition Tree

Our algorithm proceeds by finding a path of greatest weight in T_k. This can be done in linear time by depth first search. The path found in T_k represents a path in G, if we interpret paths through cycle vertices as follows. Consider a path in T_k through a cycle vertex $\langle C, u\rangle$. Both neighbours are singleton nodes, so we consider the subpath $\langle u\rangle\langle C, u\rangle\langle v\rangle$. By construction, v is connected to some vertex $w \in C$ with $w \neq u$. One of the two paths from u to w in C must have length at least half the length of C, call it P. We will interpret the path $\langle u\rangle\langle C, u\rangle\langle v\rangle$ in T_k as a path uPv in G. If a path ends in a cycle node $\langle C, u\rangle$, we may associate it with a path of length $l(C) - 1$, by moving along C from u in any of its two directions. Thus a path of weight m in T_k from the root to a leaf identifies a path of length at least m in G.

We need to show that T_k for some small k has a path of sufficient length:[1]

Lemma 3 *If G contains a path of length $r > 2^8$ starting in v then $T_k = T_k(G, v)$ for*

$$k = \lceil \tfrac{1}{4} \log r \rceil$$

contains a weighted path of length at least $\frac{1}{2}k^2 / \log\log r$.

Proof. We follow the construction of T_k in §2.2.1.

We need some additional notation. For a node $x = \langle w\rangle$ or $x = \langle C, w\rangle$ in T_k we let $L(x)$ denote the length of the longest path from w in the component $G[X]$ corresponding to the subtree rooted at x. More precisely, for every successor y of x (including $y = x$), the set X contains the corresponding vertices w' (if $y = \langle w'\rangle$ is a singleton node) or C' (if $y = \langle w', C'\rangle$ is a cycle node).

Furthermore, let $\mathbf{S}(n)$ denote the singleton node children of a node n and let $\mathbf{C}(n)$ denote its cycle node children. Consider any singleton node $\langle v\rangle$.

Lemma 2 asserts that

$$L(v) \leq \max\left\{ \max_{w \in \mathbf{S}\langle v\rangle} L(w) + k, \max_{\substack{\langle C, v\rangle \in \mathbf{C}\langle v\rangle \\ w \in \mathbf{S}\langle C, v\rangle}} \big(2L(w) + 2\big) l(C) \right\}. \tag{2.1}$$

Define $n(v) = w$ if $\langle w\rangle$ maximises the right hand side of the inequality (2.1) and consider a path $Q = \langle x_0\rangle \cdots \langle x_t\rangle$ from $\langle v\rangle = \langle x_0\rangle$ described by these heavy nodes. To be precise we have either $n(x_i) = x_{i+1}$ or $n(x_i) = x_{i+2}$, in the latter case the predecessor of $\langle x_{i+2}\rangle$ is a cycle node.

We will argue that the gaps in the sequence

$$L(x_0) \geq L(x_1) \geq \cdots \geq L(x_t).$$

cannot be too large due to the inequality above and the fact that $L(x_t)$ must be small (otherwise we are done), and therefore Q contains a lot of cycle nodes or even more singleton nodes.

Let s denote the number of cycle nodes on Q. Since every cycle node has weight at least $\frac{1}{2}k$ the total weight of Q is at least $\frac{1}{2}sk + (t - s) = s(\frac{1}{2}k - 1) + t$.

Consider a singleton node that is followed by a cycle node. There are s such nodes, we will call them *cycle parents*. Assume $\langle x_j\rangle$ is the first cycle parent node.

[1] All logarithms are to the base 2 and the constants involved have been chosen aiming for simplicity of the proof, rather than optimality.

Thus according to the first part of Lemma 2 its predecessors $\langle x_0\rangle, \ldots, \langle x_j\rangle$ satisfy the relation $L(x_{i+1}) \geq L(x_i) - k$, so

$$L(x_j) \geq r - jk \geq r - \tfrac{1}{6}k^3 \geq \tfrac{5}{6}r,$$

since $j \leq t \leq \frac{1}{6}k^2$ (otherwise we are finished) and $r \geq k^3$.

From the second part of Lemma 2 we have

$$L(x_{j+2}) \geq \frac{5r}{12l(C)} - 1 \geq \frac{2r}{k^2}.$$

where we have used $l(C) \leq \frac{1}{6}k^2$ (otherwise we are finished using the cycle as our path) and $r > 2k^2$.

This analysis may be repeated for the subsequent cycle parents as long as their remaining length after each cycle node passage is at least k^3. Note that Q must pass through as many as $s' \geq \lceil \frac{k}{\log\log r} \rceil$ cycle nodes before

$$\frac{2^{s'}r}{k^{2s'}} < k^3,$$

at which point the remaining path may be shorter than k^3. Thus we either have visited $s \geq s'$ cycle nodes, amounting to a weighted path Q of length at least

$$s(\tfrac{1}{2}k + 1) > \frac{k^2}{2\log\log r}$$

or there are at most $s < s'$ cycle nodes on Q. In that case there is a tail of singleton nodes starting with some $L(x) \geq k^3$. Since $L(x_j) \leq L(x_{j+1}) + k$ for the nodes on the tail, the length of the tail (and thus the weight of Q) is at least k^2. ∎

2.2.3 Summary

It remains to check that the path found by our algorithm satisfies the stated approximation bound: For the right k, the preceding lemma guarantees a weighted path in $T_k(G, v)$, and hence a path in G, of length

$$\frac{k^2}{2\log\log r} = \Omega\left(\frac{\log^2 r}{\log\log r}\right) = \Omega\left(\frac{\log^2 L}{\log\log L}\right)$$

because $r \geq \frac{1}{2}L$ by Lem. 1. This finishes the proof of Thm. 2.

2.3 Finding Long Paths in Hamiltonian Digraphs

Vishwanathan [22] presents a polynomial time algorithm that finds a path of length $\Omega(\log^2 n/\log\log n)$ in *undirected* Hamiltonian graphs with constant bounded degree. We show in this section that, after a minor extension to the argument, the algorithm and its analysis apply to the directed case as well.

Theorem 4 *There is a polynomial time algorithm always finding a path of length* $\Omega(\log^2 n/\log\log n)$ *in any Hamiltonian digraph of constant bounded outdegree on* n *vertices.*

To prove the theorem, we need some additional notation. Let $G = (V, A)$ be a digraph. We say that a vertex $v \in V$ *spans* the subgraph $G_v = G[V_v]$ where $V_v \subseteq V$ is the set of vertices reachable from v in G. Consider the algorithm below. It takes a digraph $G = (V, A)$ on $n = |V|$ vertices and a specified vertex $v \in V$ as input, and returns a long path starting in v.

1. Enumerate all paths in G starting in v of length $\log n$, if none return the longest found.
2. For each such path $P = (v, \cdots, w)$, let V_w be the set of vertices reachable from w in $G[V - P + \{w\}]$.
3. Compute a depth first search tree rooted at w in $G[V_w]$.
4. If the deepest path in the tree is longer than $\log^2 n$, return this path.
5. Otherwise, select the enumerated path P whose end vertex w spans as large a subgraph as possible after removal of $P - \{w\}$ from the vertex set, i.e the path maximising $|V_w|$.
6. Search recursively for a long path R starting from w in $G[V_w]$, and return $(P - \{w\}) + R$.

First note that the algorithm indeed runs in polynomial time. The enumeration of all paths of length $\log n$ takes no more than polynomial time since the outdegree is bounded by a constant k, and thus there cannot be more than $k^{\log n}$ paths. Computing a depth first search tree is also a polynomial time task, and it is seen to be performed a polynomial number of times, since the recursion does not branch at all.

To prove that the length of the resulting path is indeed $\Omega(\log^2 n / \log \log n)$, we need to show that at each recursive call of the algorithm, there is still a long enough path starting at the current root vertex.

Lemma 4 *Let $G = (V, A)$ be a Hamiltonian digraph. Let $S \subseteq V, v \in V \setminus S$. Suppose that on removal of the vertices of S, v spans the subgraph $G_v = (V_v, A_v)$ of size t. If each vertex $w \in V_v$ is reachable from v on a path of length less than d, then there is a path of length $t/(d|S|)$ in G_v starting in v.*

Proof. Consider a Hamiltonian cycle C in G. The removal of S cuts C into at most $|S|$ paths $P_1 \cdots P_{|S|}$. Since each vertex in V lies on C, the subgraph G_v must contain at least $t/|S|$ vertices W from one of the paths, say P_j. In fact, G_v must contain a path of length $t/|S|$, since the vertex in W first encountered along P_j implies the presence in G_v of all the subsequent vertices on P_j, and these are at least $|W|$. Denote one such path by $P = p_0 \cdots p_{|W|-1}$, and let $R = r_0 \cdots r_{l-1}$ be a path from $r_0 = v$ to $r_{l-1} = p_0$, of length $l \leq d$. Set $s = |P \cap R|$ and enumerate the vertices on P from 0 to $|W| - 1$ and let $i_1 \cdots i_s$ denote the indices of vertices in $P \cap R$, in particular $i_1 = 0$. Let $i_{s+1} = |W|$. An averaging argument shows that there exists j, such that $i_{j+1} - i_j \geq |W|/s$. Let q be the index for which $r_q = p_{i_j}$. The path along R from r_0 to r_q and continuing along P from p_{i_j+1} to $p_{i_{j+1}-1}$ has the claimed length. ∎

Observe that the algorithm removes no more than $\log n$ vertices from the graph at each recursive call. Thus, at call i we have removed at most $i \log n$

vertices from the original graph; the very same vertices constituting the beginning of our long path. Lemma 4 tells us that we still are in a position were it is possible to extend the path, as long as we can argue that the current end vertex of the path we are building spans large enough a subgraph. Note that whenever we stand at a vertex v starting a long path P of length $> \log n$ in step 1 of the algorithm, the path consisting of the first $\log n$ vertices of P is one of the paths of length $\log n$ being enumerated. This is our guarantee that the subgraph investigated at the next recursive call is not all that smaller than the graph considered during the previous one. It must consist of at least $|P| - \log n$ vertices. Of course, we cannot be sure that exactly this path is chosen at step 5, but this is of no concern, since it is sufficient for our purposes to assure that there are still enough vertices reachable.

Formally, let V_i denote the vertex set of the subgraph considered at the recursive call i. In the beginning, we know that regardless of the choice of start vertex v, we span the whole graph and thus $V_0 = V$, and furthermore, that a path of length n starts in v. Combining the preceding discussion with Lem. 4, we establish the following inequality for the only non-trivial case that no path of length $\log^2 n$ is ever found during step 4 of the algorithm:

$$|V_{i+1}| > \frac{|V_i|}{i \log^3 n} - \log n$$

It is readily verified that $|V_i| > 0$ for all $i < c \log n / \log \log n$ for some constant c, which completes the proof of Theorem. 4.

Chapter 3

Inapproximability Results for Longest Directed Path

In contrast to the unclear picture of the approximability of the *Longest Path* in *undirected* graph, the situation in directed graphs is drastically better: a surprisingly simple argument shows that we cannot expect to find much longer paths than our best algorithms known to date do, since if this would be the case, we could solve all problems in NP much faster than we can today. The central question of closing the gap between the upper and the lower bound for *undirected* graphs remains open, however. Our lower bound uses a reduction to the k *Vertex Disjoint Paths* problem in digraphs. Thus there is no direct way to translate our argument to the undirected case, because that problem is known to be polynomially solvable for undirected graphs [20].

3.1 Two Vertex Disjoint Paths

Our hardness proof for approximating the longest path in a directed graph starts in a reduction from a problem known to be NP-complete for over twenty years. In the k *Vertex Disjoint Paths* problem we are given a digraph G of order $n > 2k$, and we are asked whether there exists a set of k vertex disjoint paths in G such that the ith path connects vertex $2i-1$ to vertex $2i$, for $i = 1, \ldots k$. This problem is NP-complete [8] even when $k = 2$. We need to modify this result slightly to see that it is valid even if we restrict the 'yes'-instances to be partitionable into two disjoint paths. To be precise, we define the *Two Vertex Disjoint Paths* problem (2VDP): given a digraph G of order $n \geq 4$, decide whether there exists a pair of vertex disjoint paths, one from 1 to 2 and one from 3 to 4. We study the *restricted* version of this problem (R2VDP), where the 'yes'-instances are guaranteed to contain two such paths that together exhaust all vertices of G. In other words, the graph G with the additional arcs 23 and 41 contains a Hamiltonian cycle through these arcs.

Proposition 1 *Restricted Two Vertex Disjoint Paths is NP-complete.*

The proof is an extension of the construction in [8] and can be found in §3.4. It replaces a reduction from 3-Sat by a reduction from Monotone 1-in-3-Sat,

and uses a more intricate clause gadget to guarantee the existence of two paths that cover all vertices. The modification is necessary to prove the lower bound for Longest Path even for Hamiltonian instances.

3.2 Long Paths Find Vertex Disjoint Paths

We will use instances of R2VDP to build graphs in which long paths must reveal a solution to the original problem. Given an instance $G = (V, A)$ of R2VDP, define $T_d[G]$ as a graph made up out of $m = 2^d - 1$ copies $G_1 \cdots G_m$ of G arranged in a balanced binary tree structure. For all $i < 2^{d-1}$, we say that the copies G_{2i} and G_{2i+1} are the left and right *child* of the copy G_i. The copy G_1 is the *root* of the tree, and G_i for $i \geq 2^{d-1}$ are the *leaves* of the tree. The copies of G in $T_d[G]$ are connected by additional arcs as follows. For every copy G_i having children, three arcs are added (cf. Fig. 3.1):

- One arc from 2 in G_i to 1 in G_{2i}.
- One arc from 4 in G_{2i} to 1 in G_{2i+1}.
- One arc from 4 in G_{2i+1} to 3 in G_i.

Moreover, in every leaf copy G_i ($i \geq 2^{d-1}$) we add the arc 23, and in the root G_1 we add the arc 41.

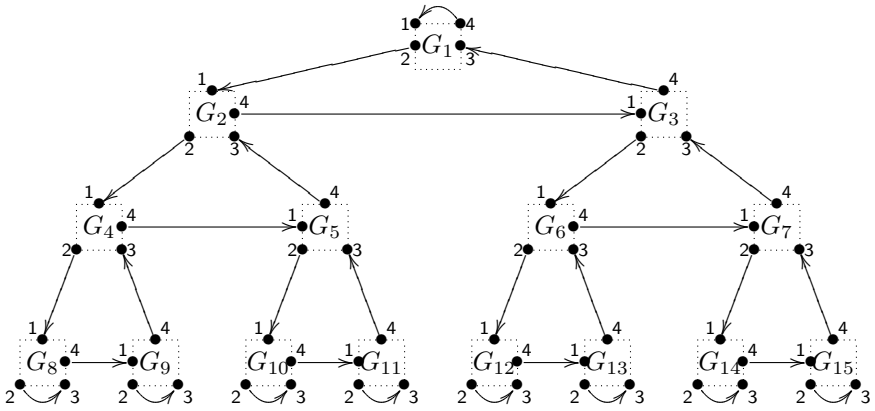

Figure 3.1: $T_4[G]$.

Lemma 5 *Given an instance $G = (V, A)$ of R2VDP on $n = |V|$ vertices, and any integers $m = 2^d - 1 > 3$, consider $T_d[G]$ with $N = mn$ vertices. Then*

- *If G has a solution then $T_d[G]$ contains a path of length $N - 1$.*
- *Given any path of length larger than $(4d - 5)n$ in $T_d[G]$, we can in time polynomial in N construct a solution to G.*

Proof. For the first part of the lemma, consider a solution for G consisting of two disjoint paths P and Q connecting 1 to 2 and 3 to 4, respectively, such that

$P + 23 + Q + 41$ is a Hamiltonian cycle in G. The copies of P and Q in all G_is together with the added arcs constitute a Hamiltonian cycle in $T_d[G]$ of length mn and thus a path of the claimed length.

For the second part, first consider an internal copy G_i and observe that if a path traverses all of the four arcs connecting G_i to the rest of the structure then this path constitutes a solution to R2VDP for G. Thus we can restrict our attention to paths in $T_d[G]$ that avoid at least one the four external arcs of each internal G_i; we call such paths *avoiding*.

Given $T_d[G]$ define $e_d[G]$ as the length of the longest avoiding path in $T_d[G]$ ending in vertex 4 of its root copy, and $s_d[G]$ as the length of the longest avoiding path starting in vertex 1 of the root copy. Consider a path P ending in vertex 4 of the root copy, for $d > 1$. At most n vertices of P are in G_1. The path P has entered G_1 via vertex 3 from G_3's vertex 4. There are two possibilities. Either the first part of P is entirely in the subtree rooted at G_3, in which case P has length at most $n + e_{d-1}[G]$. Or it entered G_3 via 1 from the subtree rooted at G_2, in which case it may pass through at most n vertices in G_3, amounting to length at most $2n + e_{d-1}[G]$. (Especially, P cannot leave via G_3's vertex 2, because then it wouldn't be avoiding). A symmetric argument for $s_d[G]$ for $d > 1$ shows an equivalent relation. Thus we have that

$$e_1[G] \leq n, \quad e_{d+1}[G] \leq 2n + e_d[G],$$
$$s_1[G] \leq n, \quad s_{d+1}[G] \leq 2n + s_d[G].$$

Furthermore, note that a longest avoiding path in $T_d[G]$ connects a path amounting to $e_{d-1}[G]$ in the right subtree, through a bridge consisting of as many vertices as possible in the root, with a path amounting to $s_{d-1}[G]$ in the left subtree. Consequently, a typical longest avoiding path starts in a leaf copy of the right subtree, travels to its sister copy, goes up a level and over to the sister of that copy, continues straight up in this zigzag manner to the root copy, and down in the same fashion on the other side. Formally, the length of a longest avoiding path in $T_d[G]$ for $d > 1$ is bounded from above by $e_{d-1}[G] + n + s_{d-1}[G] \leq (4d - 5)n$. ∎

Theorem 5 *There can be no deterministic, polynomial time approximation algorithm for Longest Path or Longest Cycle in a Hamiltonian directed graph on n vertices with performance ratio $n^{1-\epsilon}$ for any fixed $\epsilon > 0$, unless $P = NP$.*

Proof. First consider the path case. Given an instance $G = (V, A)$ of R2VDP with $n = |V|$, fix $k = 1/\epsilon$ and construct $T_d[G]$ for the smallest integers $m = 2^d - 1 \geq (4dn)^k$. Note that the graph $T_d[G]$ has order $N = n^{O(k)}$. Assume there is a deterministic algorithm finding a long path of length l_{apx} in time polynomial in N, and let l_{opt} denote the length of a longest path. Return 'yes' if and only if $l_{\text{apx}} > (4d - 5)n$. To see that this works note that if G is a 'yes'-instance and if indeed $l_{\text{opt}}/l_{\text{apx}} \leq N^{1-\epsilon}$ then $l_{\text{apx}} > (4d - 5)n$, so Lem. 5 gives a solution to G. If on the other hand G is a 'no'-instance then the longest path must be *avoiding* as defined in the proof of Lem. 5, so its length is at most $(4d - 5)n$. Thus we can solve the R2VDP problem in polynomial time, which by Prop. 1 requires P = NP.

For the cycle case, we may use a simpler construction. Simply connect m copies $G_1, \cdots, G_m$ of G on a string, by adding arcs from vertex 2 in G_i to vertex

1 in G_{i+1}, and arcs from vertex 4 in G_i to vertex 3 in G_{i-1}. Finally, add the arc 41 in G_1 and the arc 23 in G_m. The resulting graph has a cycle of length mn whenever G is a 'yes'-instance, but any cycle of size at least $2n+1$ must reveal a solution to G. ∎

3.3 Subexponential Algorithms for Satisfiability

In this section we show that a superlogarithmic dipath algorithm implies subexponential time algorithms for satisfiability.

We need the well-known reduction from Monotone 1-in-3-Sat to 3-Sat. It can be verified that the number of variables in the construction (see also [19, Exerc. 9.5.3]) is not too large:

Lemma 6 ([21]) *Given a 3-Sat instance φ with n variables and m clauses we can construct an instance of Monotone 1-in-3-Sat with $O(m)$ clauses and variables that is satisfiable if and only if φ is.*

The next lemma is a variant of Theorem 5.

Lemma 7 *There is a deterministic algorithm for Monotone 1-in-3-Sat on n variables running in time $2^{O(n^{1/(1+\epsilon)})}$, if there is*

1. *a polynomial time deterministic approximation algorithm A_{LP} for Longest Path in N-node Hamiltonian digraphs with guarantee $\log^{2+\epsilon} N$, or*
2. *a polynomial time deterministic approximation algorithm A_{LC} for Longest Cycle in N-node Hamiltonian digraphs with guarantee $\log^{1+\epsilon} N$.*

Proof. We need to verify that our constructions obey the necessary size bounds. The R2VDP instance build from the instance to Monotone 1-in-3-Sat described in §3.4 has size $n' = O(n)$.

For the path case, set $d = (4n')^{1/(1+\epsilon)}$ and construct $T_d[G]$ as in §3.2, which will have $N = (2^d - 1)n'$ nodes. Run the algorithm A_{LP} on $T_d[G]$. Observe that $(4d-5)n' < \log^{2+\epsilon}((2^d-1)n')$, so Lem. 5 tells us how to use A_{LP} to solve the R2VDP instance, and hence the 1-in-3-Sat instance.

The cycle case follows in a similar fashion from the construction in the proof of Theorem 5. ∎

Since we want to express the hardness relative to the canonical NP-complete problem 3-Sat, rather than the somewhat artificial Monotone 1-in-3-Sat, we need the Sparsification Lemma of [13] (corollary 1):

Lemma 8 (Impagliazzo, Paturi, Zane) *For all $\epsilon > 0$ and positive k, there is a constant C so that any k-Sat formula Φ with n variables, can be expressed as $\Phi = \vee_{i=1}^{t} \Psi_i$, where $t \leq 2^{\epsilon n}$ and each Ψ_i is a k-Sat formula with at most Cn clauses. Moreover, this disjunction can be computed by an algorithm running in time $poly(n)2^{\epsilon n}$.*

Theorem 6 *There is a deterministic algorithm for 3-Sat on n variables running in time $2^{o(n)}$ if there is*

1. *a polynomial time deterministic approximation algorithm for Longest Path in N-node Hamiltonian digraphs with guarantee $\log^{2+\epsilon} N$, or*

2. *a polynomial time deterministic approximation algorithm for Longest Cycle in N-node Hamiltonian digraphs with guarantee $\log^{1+\epsilon} N$.*

Proof. Use the Sparsification Lemma to write the input 3-Sat instance as a disjunction of a subexponential number of 3-Sat instances, each having linear number of clauses. Solve each of these with Lemma 6 and 7. ∎

3.4 Proof of Proposition 1

We review the construction in [8], in which the *switch* gadget from Fig. 3.2 plays a central role. Its key property is captured in the following statement.

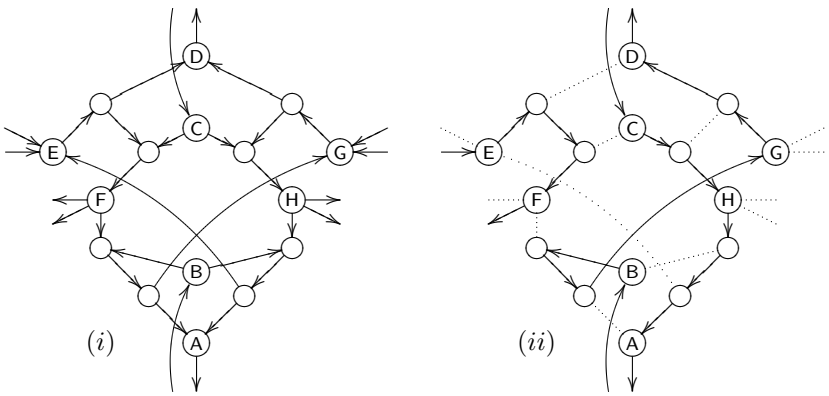

Figure 3.2: (i) A switch. Only the labelled vertices are connected to the rest of the graph, as indicated by the arrows. (ii) Three vertex-disjoint paths through a switch.

Lemma 9 ([8]) *Consider the subgraph in Fig. 3.2. Suppose that are two vertex disjoint paths passing through the subgraph—one leaving at A and the other entering at B. Then the path leaving A must have entered at C and the path entering at B must leave at D. Furthermore, there is exactly one additional path through the subgraph and it connects either E to F or G to H, depending on the actual routing of the path leaving at A.*

Also, if one of these additional paths is present, all vertices are traversed.

To prove Prop. 1 we reduce from Monotone 1-in-3-Sat, rather than 3-Sat as used in [8]. An instance of 1-in-3-Sat is a Boolean expression in conjunctive normal form in which every clause has three literals. The question is if there is a truth assignment such that in every clause, exactly one literal is true. It is known that even when all literals are positive (*Monotone* 1-in-3-Sat) the problem is NP-complete [21].

Given such an instance φ with clauses $t_1, \ldots, t_m$ on variables $x_1, \ldots, x_n$ we construct and instance G_φ of R2VDP as follows.

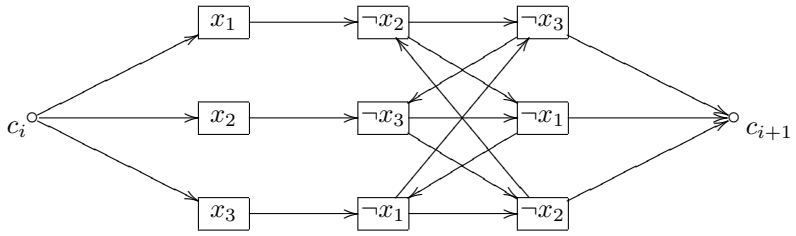

Figure 3.3: A clause gadget consisting of 9 switches. Every incoming arc to a switch ends in the switch's vertex E, and every outgoing arc leaves the switch's vertex F.

Clause gadgets. Every clause t_i is represented by a gadget consisting of a vertex c_i and nine switches, three for every literal in t_i. Consider the clause $t_i = (x_1 \vee x_2 \vee x_3)$. The vertices c_i, c_{i+1} and the E and F vertices in the nine switches are connected as shown in Fig. 3.3. Thus all clause gadgets are connected on a string ending in a dummy vertex c_{m+1}.

The clause gadget has the following desirable properties: Call a path from c_i to c_{i+1} *valid* if it is consistent with a truth assignment to $\{x_1, x_2, x_3\}$ in the sense that if it passes through a switch labelled with a literal (like $\neg x_2$) then it cannot pass through its negation (like x_2). The following claims are easily verified:

Lemma 10 *Consider the construction in Fig. 3.3.*

1. *Every valid path from c_i to c_{i+1} corresponds to a truth assignment to $\{x_1, x_2, x_3\}$ that sets exactly one variable to true.*

2. *If there is a truth assignment to $\{x_1, x_2, x_3\}$ that sets exactly one variable to true then there is a valid path from c_i to c_{i+1} corresponding to the assignment. Moreover, there is such a valid path passing through all five switches whose labels are consistent with the assignment.*

Variable gadgets. Every variable x_i is represented by a vertex v_i. (Again, vertex v_{n+1} is a dummy vertex.) All switches in the clause gadgets representing the positive literal of the variable v_i are connected in series (the ordering of the switches on this string is not important): the vertex H in a switch is connected to vertex G of the next switch with the same label. Furthermore, there is an arc from v_i to vertex G in the first switch on its literal path, and an arc from vertex H in the last switch on the path to vertex v_{i+1}.

Likewise, all switches labelled with negated literals of this variable are connected. Thus there are two strings of switches leaving v_i: one contains all the positive literals, and one contains all the negated literals. Both end in v_{i+1}.

Also, *all* the switches are arranged on a path and connected by added arcs from vertex A in a switch to vertex C in the next one, and arcs back from vertex D in a switch to vertex B of the preceding switch. The ordering of the switches on this switch path is not important.

Finally, there is an arc from v_{n+1} to c_1 and an arc from vertex D in the first switch on the switch path to v_1.

To finish the construction of an instance of R2VDP it remains to identify the first four vertices. Vertex 1 is vertex B of the last switch on the switch path, vertex 2 is c_{m+1}, vertex 3 is vertex C of the first switch on the switch path, and vertex 4 is vertex A of the last switch on the switch path.

Lemma 11 *G_φ has two vertex disjoint paths from 1 to 2 and from 3 to 4 if and only if φ has a solution. Moreover, if G_φ contains such paths then it contains two such paths that together exhaust all its vertices.*

Proof. Assume φ can be satisfied so that exactly one variable in every clause is true. Walk through G_φ starting in vertex 1. This path is forced to traverse all switches until it reaches v_1. In general, assume that we reached v_i. To continue to v_{i+1} traverse the G–H paths of the string of negative literal switches if x_i is true; otherwise take the string of positive literal switches. Note that this forces us to avoid the E–F paths in these switches later.

Arriving at v_{n+1} continue to c_1. To travel from c_i to c_{i+1} we are forced to traverse the clause gadget of Fig. 3.3. Note that the truth assignment has set exactly one of the variables to true, blocking the E–F path in the two switches labelled by its negative literal. Likewise, two of the variables are false, blocking the (two) switches labelled by their positive literal. The remaining five switches are labelled by the positive literal of the true variable or negative literals of the falsified variables. The valid path ensured by Lem. 10 passes through exactly these five switches.

Finally, the path arrives at $v_{m+1} = 2$. The path travelling from 3 to 4 is now unique. Observe that the two paths exhaust all the vertices and thus form a Hamiltonian cycle if we add 23 and 41.

Conversely, assume there are two paths from 1 to 2 and from 3 to 4. The subpaths connecting v_i to v_{i+1} ensure that all literal switches are consistent in the sense that if the E–F path in a switch labelled x_i is blocked then it is blocked in *all* such switches, and not blocked in any switch labelled $\neg x_i$. This forces the subpaths from c_i to c_{i+1} to be valid. Lem. 10 ensures that the corresponding truth assignment is satisfying and sets exactly one variable in each clause. ∎

Bibliography

[1] N. Alon, R. Yuster, and U. Zwick. Color-coding. *Journal of the ACM*, 42(4), pp. 844–856, 1995.

[2] D. Angluin and L. G. Valiant. Fast probabilistic algorithms for Hamiltonian circuits and matchings. *Journal of Computer and Systems Sciences*, 18(2), pp. 155–193, 1979. Announced at STOC '77.

[3] A. Björklund and T. Husfeldt. Finding long paths in directed graphs is hard. Manuscript, 2001.

[4] H. L. Bodlaender. On linear time minor tests with depth-first search. *Journal of Algorithms*, 14(1), pp. 1–23, 1993.

[5] J. A. Bondy and S. C. Locke. Relative length of paths and cycles in 3-connected graphs. *Discrete Mathematics*, 33, pp. 111–122, 1981.

[6] A. Z. Broder, A. M. Frieze, and E. Shamir. Finding hidden Hamiltonian cycles. In *Proc. 23nd STOC*, pp. 182–189, ACM, 1991.

[7] T. Feder, R. Motwani, and C. S. Subi. Approximating the longest cycle problem in sparse graphs In *SIAM Journal of Computing*, 31(5), pp. 1596–1607, 2002.

[8] S. Fortune, J. Hopcroft, and J. Wyllie. The directed subgraph homeomorphism problem. In *Theoretical Computer Science*, 10, pp. 111–121, 1980.

[9] H. Gabow and S. Nie. Finding a long directed cycle. In *Proc. 15th SODA*, 2004.

[10] Y. Gurevich and S. Shelah. Expected computation time for Hamiltonian path problem. In *SIAM Journal of Computing*, 16(3), pp. 486–502, 1987.

[11] W. R. Hamilton. Memorandum respecting a new System of Roots of Unity In *The Philosophical Magazine*, volume 12 (4th series), pp. 446, 1856.

[12] R. Impagliazzo and R. Paturi. On the complexity of k-SAT. *Journal of Computer and Systems Sciences*, 62(2), pp. 367–375, 2001. Announced at CoCo '99.

[13] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? In *Proc. 39th FOCS*, pp. 653–663, 1998.

[14] D. Karger, R. Motwani, and G.D.S. Ramkumar. On approximating the longest path in a graph. *Algorithmica*, 18(1), pp. 82–98, 1997.

[15] R. M. Karp Reducibility among combinatorial problems. In R. E. Miller and J. W Thatcher (eds.), *Complexity of Computer Computations*, Plenum Press, New York, pp. 85–103, 1972.

[16] S. Khanna. Longest directed path is $n^{1-\epsilon}$-hard. Manuscript, 1999.

[17] A. Lubotzky. Discrete groups, expanding graphs and invariant measures. Birkhäuser Verlag, Basel, 1994.

[18] B. Monien. How to find long paths efficiently. *Annals of Discrete Mathematics*, 25, pp. 239–254, 1985.

[19] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[20] N. Robertson and P. D. Seymour. Graph minors XIII: The disjoints paths problem. *J. Combinatorial Theory Ser. B*, 35, 1983.

[21] T. J. Schaefer. The complexity of satisfiability problems. In *Proc. 10th STOC*, pp. 216–226, 1978.

[22] S. Vishwanathan. An approximation algorithm for finding a long path in Hamiltonian graphs. In *Proc. 11th SODA*, pp. 680–685, 2000.