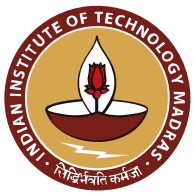


IIT Madras

Research Report

Summer Research Fellowship 2024

Written by

Albin James Maliakal
CS24SFP5773

Supervised by

Dr. Akanksha Agrawal
Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Madras

From 24/05/2024 to 15/07/2024

2024 - CSE - IIT Madras

Written at Research in Algorithms & Graphs (RAnG) Lab,
Department of Computer Science and Engineering,
Indian Institute of Technology Madras,
Chennai, Tamilnadu, India - 600036

Abstract

The report presents a rigorous exploration of advanced algorithmic topics, focusing on the derivation and validation of proofs across various domains. The report begins by elucidating the fundamental concepts of characterizing running times through Big Oh and Little Oh notations, providing formal definitions and detailed explanations to facilitate a deep understanding of algorithm analysis complexities.

Furthermore, the report delves into the realm of greedy algorithms, specifically examining interval scheduling as a prime example. By presenting pseudocode implementations and proofs of correctness, the document showcases the application of greedy strategies in algorithm design, emphasizing the importance of efficiency and optimality in computational problem-solving.

Moreover, it delves into intricate algorithmic techniques such as Baker's Technique, offering an approximation algorithm for planar graphs and an exact algorithm for outerplanar graphs. Through detailed explanations of constructing graph representations, labeling techniques, and dynamic programming approaches, the report demonstrates the intricacies of solving complex graph-related problems with precision and accuracy.

Additionally, it explores matrix operations, particularly focusing on LUP decomposition methods. By providing insights into the overview of LUP decomposition, forward and back substitution techniques, pseudocodes for key algorithms, and proofs of correctness, the report equips readers with a comprehensive understanding of matrix manipulation algorithms crucial for various computational tasks.

Furthermore, the report delves into network flow problems, introducing the Ford-Fulkerson Algorithm as a solution methodology. By elucidating the algorithm's workings in optimizing network flow, it offers practical insights into solving real-world problems using advanced algorithmic techniques, highlighting the significance of algorithmic efficiency in addressing complex computational challenges.

In essence, it serves as a valuable resource for individuals seeking a detailed exploration of algorithmic proofs, greedy strategies, graph algorithms, matrix operations, and network flow optimization. Its rigorous approach to deriving and validating proofs across diverse algorithmic domains makes it an essential read for students, researchers, and professionals in the field of computer science and algorithm design.

Acknowledgments

I extend my heartfelt appreciation to Dr. Akanksha Agrawal, Assistant Professor at the Department of Computer Science and Engineering, Indian Institute of Technology Madras, for her exceptional mentorship and unwavering support during the Summer Research Fellowship 2024. Her expertise and guidance have been instrumental in shaping my research journey and enhancing my understanding of complex algorithms and graph theory.

I am deeply grateful to Indian Institute of Technology Madras for granting me the opportunity to partake in this enriching experience at the RAnG Lab. The conducive research environment and resources provided by the institute have been pivotal in fostering my academic growth and research skills. The exposure to cutting-edge research in Algorithms & Graphs has been both enlightening and inspiring, fueling my passion for computational problem-solving.

This Summer Research Fellowship has not only broadened my knowledge but has also instilled in me a deeper appreciation for the intricacies of computer science research. I am thankful for the support, encouragement, and intellectual stimulation that I have received from both Prof. Akanksha Agrawal and Prof. N. S. Narayanaswamy. This experience has been transformative, equipping me with invaluable insights and skills that will undoubtedly shape my future endeavors in the field of Computer Science and Engineering.

Contents

1 Characterizing Running Times

Contents

1.1 Big Oh Notation

Big Oh notation, denoted as $O(g(n))$, characterizes an upper bound on the asymptotic behavior of a function. It signifies that a function does not grow faster than a certain rate determined by the highest-order term $g(n)$. For example, if we have a function $f(n) = 7n^3 + 100n^2 + 20n + 6$, and we observe that its highest-order term is $7n^3$, we can say that the function's growth rate is n^3. Therefore, we can represent this function as $O(n^3)$. It's important to note that if a function grows no faster than a certain rate, we can use Big Oh notation to describe its growth rate [1].

1.1.1 Formal Definition of Big Oh Notation

The formal definition of Big Oh notation is given by: $O(g(n)) = \{f(n) \mid$ there exist positive constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0\}$

1.2 Little Oh Notation

Little Oh notation, represented as $o(g(n))$, is used to denote an upper bound that is not asymptotically tight. It signifies that a function becomes insignificant relative to $g(n)$ as n grows large [1].

1.2.1 Formal Definition of Little Oh Notation

The formal definition of Little Oh notation is given by: $o(g(n)) = \{f(n) \mid$ for any positive constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < c \cdot g(n)$ for all $n \geq n_0\}$

2

2 Greedy Algorithms

Contents

2.1 Interval Scheduling

The Interval Scheduling Problem involves selecting requests to maximize the number of accepted intervals without overlaps. Greedy algorithms for this problem aim to select requests based on certain rules to achieve an optimal solution. Several natural rules have been explored:

1. **Selecting the earliest start time:** Initially, choosing the request with the earliest start time may seem intuitive, but it can lead to suboptimal solutions, especially if the selected request is for a long interval, potentially causing the rejection of shorter requests.

2. **Selecting the shortest interval:** Another approach is to select the request with the smallest interval, minimizing the time the resource is occupied. While this rule is better than selecting the earliest start time, it can still result in suboptimal schedules by preventing the acceptance of other requests.

3. **Considering conflicts:** A more refined greedy rule involves selecting the request with the fewest conflicts with other requests. By minimizing noncompatible requests, this approach can lead to optimal solutions in many cases, ensuring efficient resource utilization.

4. **Accepting the request that finishes first:** A successful greedy rule is to prioritize requests based on their finish times, accepting the request that finishes earliest. This strategy aims to free up the resource quickly while satisfying requests, potentially maximizing the number of accommodated intervals.

These rules demonstrate the iterative nature of greedy algorithms in interval scheduling, where each step involves selecting a request and rejecting incompatible ones to build an optimal solution.

2.1.1 Pseudocode

Algorithm 1 Interval Scheduling Algorithm

```
Let R be the set of all requests
Let A be empty
while R is not empty do
    Choose a request i ∈ R that has the smallest finishing time
    Add request i to A
    Delete all requests from R that are not compatible with request i
end while
Return the set A as the set of accepted requests
```

2.1.2 Proof of Correctness:

Proof of (4.2)

Claim: For all indices $r \leq k$, we have $f(i_r) \leq f(j_r)$.

Proof: We will prove this statement by induction.

Base Case: For $r = 1$, the statement is clearly true as the algorithm starts by selecting the request i_1 with the minimum finish time.

Inductive Step: Now, let $r > 1$. Assume as our induction hypothesis that the statement is true for $r - 1$, and we will try to prove it for r. Let's assume $f(i_{r-1}) \leq f(j_{r-1})$. In order for the algorithm's r-th interval not to finish earlier as well, it would need to "fall behind." However, the greedy algorithm always has the option (at worst) of choosing j_r and thus fulfilling the induction step.

We know (since O consists of compatible intervals) that $f(j_{r-1}) \leq s(j_r)$. Combining this with the induction hypothesis $f(i_{r-1}) \leq f(j_{r-1})$, we get $f(i_{r-1}) \leq s(j_r)$. Thus, the interval j_r is in the set R of available intervals at the time when the greedy algorithm selects i_r. The greedy algorithm selects the available interval with the smallest finish time; since interval j_r is one of these available intervals, we have $f(i_r) \leq f(j_r)$. This completes the induction step.

Therefore, the greedy algorithm "stays ahead" of O as each interval it selects finishes at least as soon as the corresponding interval in O.

Proof of (4.3)

Claim: The greedy algorithm returns an optimal set A.

Proof: We will prove the statement by contradiction. If A is not optimal, then an optimal set O must have more requests, i.e., $m > k$. Applying (4.2) with $r = k$, we get that $f(i_k) \leq f(j_k)$. Since $m > k$, there is a request j_{k+1} in O. This request starts after request j_k ends, and hence after i_k ends. So after deleting all requests that are not compatible with requests $i_1, \ldots, i_k$, the set of possible requests R still contains j_{k+1}. But the greedy algorithm stops with request i_k, and it is only supposed to stop when R is empty—a contradiction.

2.1.3 Complexity Analysis:

The sorting step dominates with $O(n \log n)$ time complexity, where n is the number of intervals. The greedy selection process takes $O(n)$ time. Overall complexity is $O(n \log n)$.

2.2 Djikstra's Algorithm

Initialization:

Maintain a set S of vertices u for which the shortest-path distance $d(u)$ from s has been determined. Initially, $S = \{s\}$ and $d(s) = 0$.

Main Steps:

- For each node $v \in V - S$ (nodes not yet explored), determine the shortest path that can be constructed by traveling along a path through the explored part S to some $u \in S$, followed by the single edge (u, v).
- Define a new distance metric $d'(v) = \min_{e=(u,v):u \in S}(d(u) + w(e))$, where $w(e)$ represents the weight of edge e.
- Choose the node $v \in V - S$ for which $d'(v)$ is minimized.
- Add node v to S and define $d(v)$ to be the value of $d'(v)$.

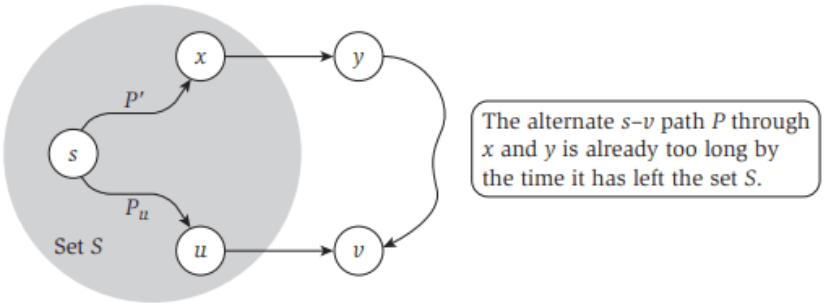


Figure 4.8 The shortest path P_v and an alternate s-v path P through the node y.

Result:

The algorithm iterates this process until all nodes have been explored (i.e., $S = V$), finding the shortest paths from the start node s to every other node in the graph.

2.2.1 Pseudocode

Algorithm 2 Dijkstra's Algorithm

```
Let S be the set of explored nodes
for each u ∈ S do
    Store a distance d(u)
end for
Initially S = {s} and d(s) = 0
while S ≠ V do
    Select a node v ∈ S with at least one edge from S for which
    d'(v) = min_{e=(u,v):u∈S}(d(u) + w(e)) is as small as possible
    Add v to S and define d(v) = d'(v)
end while
```

2.2.2 Proof of Correctness of Dijkstra's Algorithm

Claim: For each $u \in S$ at any point in the algorithm's execution, the path P_u is a shortest $s - u$ path.

Proof: We prove this by induction on the size of S.

Base Case: When $|S| = 1$, we have $S = \{s\}$ and $d(s) = 0$, which is trivially the shortest path from s to s.

Inductive Step: Suppose the claim holds when $|S| = k$ for some $k \geq 1$. Now, we grow S to size $k + 1$ by adding the node v. Let (u, v) be the final edge on our $s - v$ path P_v.

By the induction hypothesis, P_u is the shortest $s-u$ path for each $u \in S$. Consider any other $s-v$ path P; we wish to show that it is at least as long as P_v.

Let y be the first node on P that is not in S, and let $x \in S$ be the node just before y. The situation is as depicted in Figure 4.8.

The crux of the proof is that P cannot be shorter than P_v by the time it has left the set S. In iteration $k+1$, Dijkstra's Algorithm must have considered adding node y to the set S via the edge (x, y) and rejected this option in favor of adding v. This implies that there is no path from s to y through x that is shorter than P_v.

Since edge lengths are nonnegative, the full path P is at least as long as P_v as well.

This completes the proof.

Additional Inequalities: Let P' be the subpath of P from s to x. Since $x \in S$, by the induction hypothesis, P_x is a shortest $s-x$ path of length $d(x)$. Thus, the subpath of P out to node y has length $\text{len}(P') + \text{len}(x, y) \geq d(x) + \text{len}(x, y) \geq d'(y)$, and the full path P is at least as long as this subpath. Finally, since Dijkstra's Algorithm selected v in this iteration, we know that $d'(y) \geq d'(v) = \text{len}(P_v)$. Combining these inequalities shows that $\text{len}(P) \geq \text{len}(P') + \text{len}(x, y) \geq \text{len}(P_v)$.

2.2.3 Complexity Analysis of Dijkstra's Algorithm

Let n be the number of nodes in the graph and m be the number of edges.

Time Complexity:

The time complexity of Dijkstra's algorithm using a priority queue is $O((n+m)\log n)$.

Explanation:

- Initializing the distance array and priority queue takes $O(n \log n)$ time.
- Each edge is relaxed at most once, which takes $O(\log n)$ time in the priority queue.
- Overall, the time complexity is dominated by the edge relaxation step, resulting in $O((n+m)\log n)$.

Space Complexity:

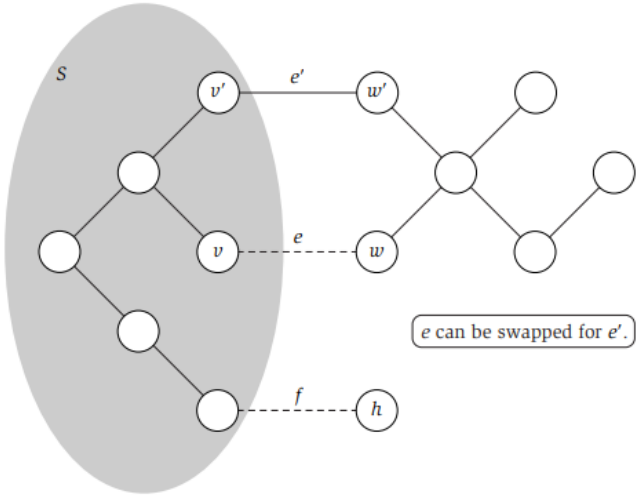


Figure 4.10 Swapping the edge e for the edge e' in the spanning tree T, as described in the proof of (4.17).

The space complexity of Dijkstra's algorithm using a priority queue is $O(n)$.

Explanation:

- The priority queue and distance array both require $O(n)$ space.
- Additional space complexity is negligible compared to the input size.

2.3 Minimum Spanning Tree

2.3.1 Proof of Correctness of Kruskal's Algorithm

Theorem: Kruskal's Algorithm produces a minimum spanning tree of G.

Proof: Consider any edge $e = (v, w)$ added by Kruskal's Algorithm, and let S be the set of all nodes to which v has a path at the moment just before e is added. Clearly $v \in S$, but $w \notin S$, since adding e does not create a cycle. Moreover, no edge from S to $V - S$ has been encountered yet, since any such edge could have been added without creating a cycle, and hence would have been added by Kruskal's Algorithm. Thus, e is the cheapest edge with one end in S and the other in $V - S$, and so it belongs to every minimum spanning tree.

To complete the proof, we need to show that the output (V, T) of Kruskal's Algorithm is indeed a spanning tree of G. Clearly, (V, T) contains no cycles, since the algorithm is explicitly designed to avoid creating cycles. Furthermore, if (V, T) were not connected, then there would exist a

nonempty subset of nodes S (not equal to all of V) such that there is no edge from S to $V - S$. However, this contradicts the behavior of the algorithm: since G is connected, there is at least one edge between S and $V - S$, and the algorithm will add the first of these that it encounters.

This completes the proof.

2.3.2 Proof of Correctness of Prim's Algorithm

Theorem: Prim's Algorithm produces a minimum spanning tree of G.

Proof: For Prim's Algorithm, it is also very easy to show that it only adds edges belonging to every minimum spanning tree. Indeed, in each iteration of the algorithm, there is a set $S \subseteq V$ on which a partial spanning tree has been constructed, and a node v and edge e are added that minimize the quantity $\min_{e=(u,v):u\in S} c_e$. By definition, e is the cheapest edge with one end in S and the other end in $V - S$, and so by the Cut Property it is in every minimum spanning tree.

It is also straightforward to show that Prim's Algorithm produces a spanning tree of G, and hence it produces a minimum spanning tree.

3 Network Flow

Contents

3.1 Ford-Fulkerson Algorithm

The Ford-Fulkerson Algorithm is a method for computing the maximum flow in a flow network. It operates by iteratively finding augmenting paths from the source to the sink in the residual graph, increasing the flow along these paths until no more augmenting paths can be found. The algorithm terminates when no more augmenting paths exist, at which point the maximum flow is achieved.

3.1.1 Pseudocode

Algorithm 3 Ford-Fulkerson Algorithm

```
1: procedure FordFulkerson(Graph G = (V, E), Source s, Sink t)
2:     Initialize flow f to 0 on all edges
3:     while there exists an augmenting path p in the residual graph do
4:         Find the bottleneck capacity c_f of path p
5:         for each edge e in path p do
6:             if e is a forward edge then
7:                 Increase flow f(e) by c_f
8:             else
9:                 Decrease flow f(e) by c_f
10:            end if
11:        end for
12:    end while
13:    return Maximum flow f
14: end procedure
```

3.1.2 Proof of Correctness

Claim: If f is an $s-t$ flow such that there is no $s-t$ path in the residual graph G_f, then there is an $s-t$ cut (A^*, B^*) in G for which $\nu(f) = c(A^*, B^*)$. Consequently, f has the maximum value of any flow in G, and (A^*, B^*) has the minimum capacity of any $s-t$ cut in G.

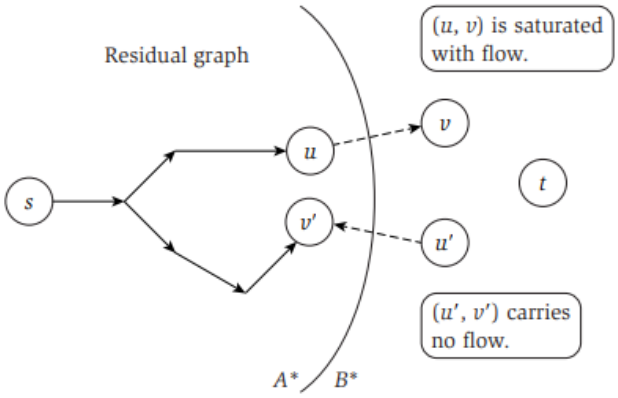


Figure 7.5 The (A^*, B^*) cut in the proof of (7.9).

Proof: The statement claims the existence of a cut satisfying a certain desirable property; thus we must now identify such a cut. To this end, let A^* denote the set of all nodes v in G for which there is an $s - v$ path in G_f. Let B^* denote the set of all other nodes: $B^* = V - A^*$.

First we establish that (A^*, B^*) is indeed an $s-t$ cut. It is clearly a partition of V. The source s belongs to A^* since there is always a path from s to s. Moreover, $t \notin A^*$ by the assumption that there is no $s - t$ path in the residual graph; hence $t \in B^*$ as desired.

Next, suppose that $e = (u, v)$ is an edge in G for which $u \in A^*$ and $v \in B^*$, as shown in Figure 7.5. We claim that $f(e) = c_e$. For if not, e would be a forward edge in the residual graph G_f, and since $u \in A^*$, there is an $s - u$ path in G_f; appending e to this path, we would obtain an $s - v$ path in G_f, contradicting our assumption that $v \in B^*$.

Now suppose that $e' = (u', v')$ is an edge in G for which $u' \in B^*$ and $v' \in A^*$. We claim that $f(e') = 0$. For if not, e' would give rise to a backward edge $e'' = (v', u')$ in the residual graph G_f, and since $v' \in A^*$, there is an $s - v'$ path in G_f; appending e'' to this path, we would obtain an $s - u'$ path in G_f, contradicting our assumption that $u' \in B^*$.

So all edges out of A^* are completely saturated with flow, while all edges into A^* are completely unused. We can now use (7.6) to reach the desired conclusion:

$$\begin{aligned}
\nu(f) &= f_{\text{out}}(A^*) - f_{\text{in}}(A^*) \\
&= \sum_{e \text{ out of } A^*} f(e) - \sum_{e \text{ into } A^*} f(e) \\
&= \sum_{e \text{ out of } A^*} c_e - 0 \\
&= c(A^*, B^*).
\end{aligned}$$

3.1.3 Complexity Analysis

Let n denote the number of nodes in the flow network G, and m denote the number of edges in G. We have assumed that all nodes have at least one incident edge, hence $m \geq n/2$, and so we can use $O(m+n) = O(m)$ to simplify the bounds.

Claim: Suppose all capacities in the flow network G are integers. Then the Ford-Fulkerson Algorithm can be implemented to run in $O(mC)$ time, where C is the maximum capacity of any edge leaving the source node s.

Proof: We know from a previous claim that the algorithm terminates in at most C iterations of the `While` loop. We therefore consider the amount of work involved in one iteration when the current flow is f.

The residual graph G_f has at most $2m$ edges, since each edge of G gives rise to at most two edges in the residual graph. We will maintain G_f using an adjacency list representation; we will have two linked lists for each node v, one containing the edges entering v, and one containing the edges leaving v.

To find an $s-t$ path in G_f, we can use breadth-first search or depth-first search, which can be implemented to run in $O(m)$ time. Augmenting the flow along this path can also be done in $O(m)$ time.

Therefore, each iteration of the `While` loop takes $O(m)$ time. Since there are at most C iterations, the total running time of the Ford-Fulkerson Algorithm is $O(mC)$.

4 Multiplication using FFT

Contents

4.1 Introduction to FFT Multiplication

To break through the quadratic time barrier for convolutions, we are going to exploit the connection between the convolution and the multiplication of two polynomials. But rather than use convolution as a primitive in polynomial multiplication, we are going to exploit this connection in the opposite direction. Suppose we are given the vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$. We will view them as the polynomials $A(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_{n-1} x^{n-1}$ and $B(x) = b_0 + b_1 x + b_2 x^2 + \ldots + b_{n-1} x^{n-1}$, and we'll seek to compute their product $C(x) = A(x)B(x)$ in $O(n \log n)$ time. If $c = (c_0, c_1, \ldots, c_{2n-2})$ is the vector of coefficients of C, then c is exactly the convolution $a * b$, and so we can then read off the desired answer directly from the coefficients of $C(x)$.

Now, rather than multiplying A and B symbolically, we can treat them as functions of the variable x and multiply them as follows:

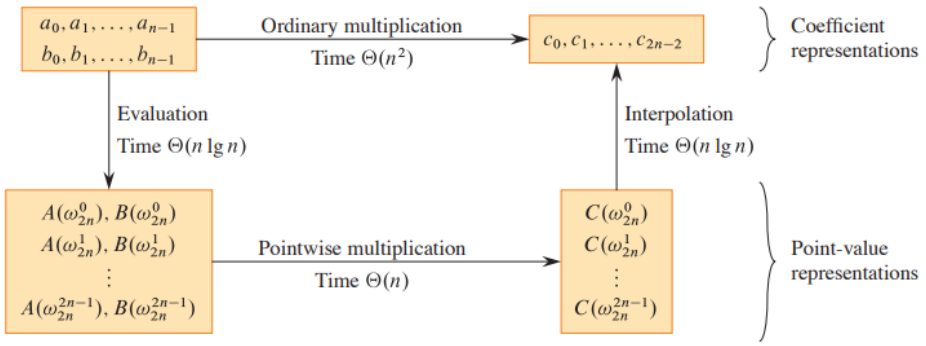
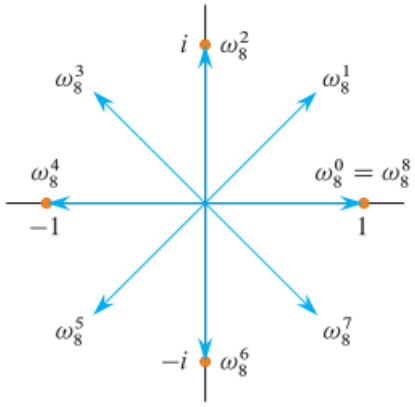


Figure 30.1 A graphical outline of an efficient polynomial-multiplication process. Representations on the top are in coefficient form, and those on the bottom are in point-value form. The arrows from left to right correspond to the multiplication operation. The ω_{2n} terms are complex $(2n)$th roots of unity.



1. First we choose $2n$ values $x_1, x_2, \ldots, x_{2n}$ and evaluate $A(x_j)$ and $B(x_j)$ for each of $j = 1, 2, \ldots, 2n$.

2. We can now compute $C(x_j)$ for each j very easily: $C(x_j)$ is simply the product of the two numbers $A(x_j)$ and $B(x_j)$.

3. Finally, we have to recover C from its values on $x_1, x_2, \ldots, x_{2n}$. Here we take advantage of a fundamental fact about polynomials: any polynomial of degree d can be reconstructed from its values on any set of $d+1$ or more points. This is known as polynomial interpolation. We simply observe that since A and B each have degree at most $n-1$, their product C has degree at most $2n-2$, and so it can be reconstructed from the values $C(x_1), C(x_2), \ldots, C(x_{2n})$ that we computed in step (ii).

This approach to multiplying polynomials has some promising aspects and some problematic ones. First, the good news: step (ii) requires only $O(n)$ arithmetic operations, since it simply involves the multiplication of $O(n)$ numbers. But the situation doesn't look as hopeful with steps (i) and (iii). In particular, evaluating the polynomials A and B on a single value takes $O(n)$ operations, and our plan calls for performing $2n$ such evaluations. This seems to bring us back to quadratic time right away.

The key idea that will make this all work is to find a set of $2n$ values $x_1, x_2, \ldots, x_{2n}$ that are intimately related in some way, such that the work in evaluating A and B on all of them can be shared across different evaluations. A set for which this will turn out to work very well is the complex roots of unity.

The Complex Roots of Unity: The complex numbers can be viewed as lying in the "complex plane," with axes representing their real and imaginary parts. We can write a complex number

using polar coordinates with respect to this plane as $re^{\theta i}$, where $e^{\pi i} = -1$ (and $e^{2\pi i} = 1$). Now, for a positive integer k, the polynomial equation $x^k = 1$ has k distinct complex roots, and it is easy to identify them. Each of the complex numbers $\omega_{j,k} = e^{2\pi ji/k}$ (for $j = 0, 1, 2, \ldots, k-1$) satisfies the equation, since $(e^{2\pi ji/k})^k = e^{2\pi ji} = (e^{2\pi i})^j = 1^j = 1$, and each of these numbers is distinct, so these are all the roots. We refer to these numbers as the kth roots of unity. We can picture these roots as a set of k equally spaced points lying on the unit circle in the complex plane.

For our numbers $x_1, \ldots, x_{2n}$ on which to evaluate A and B, we will choose the $(2n)$th roots of unity. It's worth mentioning (although it's not necessary for understanding the algorithm) that the use of the complex roots of unity is the basis for the name Fast Fourier Transform: the representation of a degree-d polynomial P by its values on the $(d+1)$st roots of unity is sometimes referred to as the discrete Fourier transform of P; and the heart of our procedure is a method for making this computation fast [2].

4.2 Pseudocode

Multiplication of two n-bit numbers using Fast Fourier Transform (FFT) involves representing the numbers as polynomials and leveraging the FFT algorithm to compute their product efficiently. The process includes extending the polynomials to degree $2n-2$, applying FFT to transform them into the frequency domain, performing pointwise multiplication, and then applying the Inverse FFT to obtain the final result. This algorithm achieves a time complexity of $O(n \log n)$, making it significantly faster than traditional multiplication methods for large numbers.

4.3 Proof of Correctness

Base Case:

- For $n = 1$, the algorithm performs direct multiplication of the coefficients of A and B, which is trivially correct.

Inductive Hypothesis:

- Assume that the FFT multiplication algorithm correctly computes the product of two $(n-1)$-bit numbers for all $n \geq 1$.

Algorithm 4 FFT Multiplication

```
function FFT_Multiply(A, B)
    n ← length(A)
    m ← 2 × n − 1                                   ▷ Extend to degree 2n − 2
    A′ ← extend_polynomial(A, m)
    B′ ← extend_polynomial(B, m)
    A_hat ← FFT(A′)
    B_hat ← FFT(B′)
    C_hat ← A_hat × B_hat
    C ← Inverse_FFT(C_hat)
    return C[0 : n − 1]                             ▷ Extract coefficients for product of A and B
end function
function extend_polynomial(P, m)
    n ← length(P)
    P_extended ← new array of size m filled with zeros
    for i ← 0 to n − 1 do
        P_extended[i] ← P[i]
    end for
    return P_extended
end function
```

Inductive Step:

- Consider two n-bit numbers A and B represented as polynomials $A(x)$ and $B(x)$ respectively.
- Divide A and B into two $(n/2)$-bit numbers each: $A = A_0 + A_1 x^{n/2}$ and $B = B_0 + B_1 x^{n/2}$.
- Recursively apply the FFT multiplication algorithm to compute the products $A_0 \cdot B_0$, $A_1 \cdot B_1$, and $(A_0 + A_1) \cdot (B_0 + B_1)$.
- Combine the results using FFT to obtain the product polynomial $C(x) = A(x) \cdot B(x)$.

Correctness Verification:

- By the inductive hypothesis, the algorithm correctly computes the products of $(n/2)$-bit numbers.
- The combination step using FFT ensures that the product of the n-bit numbers A and B is computed accurately.
- The properties of FFT guarantee that the convolution operation in the frequency domain yields the correct product polynomial.

Conclusion:

- Therefore, by induction, the FFT multiplication algorithm correctly computes the product of two n-bit numbers represented as polynomials, ensuring the correctness of the algorithm.

By leveraging the properties of the DFT and the inverse DFT, the FFT-based multiplication algorithm correctly computes the coefficients c_k of the product polynomial $C(x)$. These coefficients correspond to the product of the two original numbers A and B when interpreted in the polynomial basis. Thus, the FFT-based multiplication algorithm is mathematically correct for multiplying two n-bit numbers.

4.4 Complexity of FFT Multiplication Algorithm

Let n be the number of bits in each of the two numbers being multiplied.

Step 1: FFT Computation

Computing the Fast Fourier Transform (FFT) of each of the two input polynomials requires $O(n \log n)$ operations for each polynomial. Therefore, the total complexity for FFT computation is $O(n \log n)$ for both polynomials.

Step 2: Pointwise Multiplication

The pointwise multiplication of the two FFTs requires $O(n)$ operations, as each coefficient of the polynomials is multiplied with its corresponding coefficient in the other polynomial.

Step 3: Inverse FFT

Computing the Inverse FFT of the product polynomial also requires $O(n \log n)$ operations.

Overall Complexity

Adding up the complexities of all steps, the overall complexity of the FFT Multiplication algorithm for two n-bit numbers is $O(n \log n)$.

5 Randomized Algorithms

Contents

5.1 Finding the Global Minimum Cut

The Contraction Algorithm, operates on a connected multigraph $G = (V, E)$ where multiple parallel edges between nodes are allowed. The algorithm randomly selects an edge $e = (u, v)$ from G and contracts it, merging nodes u and v into a new node w in the resulting graph G'. This process eliminates edges between u and v while updating other edges in G' accordingly. Despite G having at most one edge between any pair of nodes, G' may contain parallel edges.

Subsequently, the Contraction Algorithm recursively operates on G', selecting edges randomly for contraction. During these recursive steps, the vertices of G' are considered as supernodes, where each supernode w corresponds to a subset $S(w) \subseteq V$ absorbed during the contractions. The algorithm concludes when G' reduces to two supernodes v_1 and v_2, potentially with parallel edges between them. Each supernode v_i is associated with a subset $S(v_i) \subseteq V$ comprising the nodes absorbed during the contraction process [2].

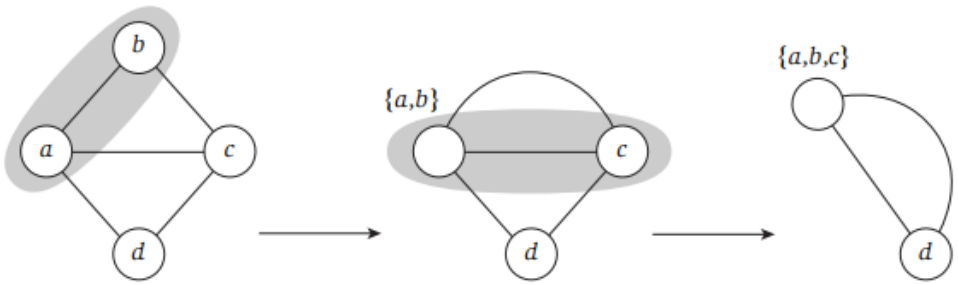


Figure 13.1 The Contraction Algorithm applied to a four-node input graph.

Algorithm 5 Contraction Algorithm for Multigraph $G = (V, E)$

for each node v **do**
    Record the set $S(v)$ of nodes that have been contracted into v
    Initially $S(v) = \{v\}$ for each v
end for
if G has two nodes v_1 and v_2 **then**
    return the cut $(S(v_1), S(v_2))$
else
    Choose an edge $e = (u, v)$ of G uniformly at random
    Let G' be the graph resulting from the contraction of e, with a new node z_{uv} replacing u and v
    Define $S(z_{uv}) = S(u) \cup S(v)$
    Apply the Contraction Algorithm recursively to G'
end if

5.1.1 Pseudocode

5.1.2 Proof of Correctness

(13.4) There is a polynomial-time algorithm to find a global min-cut in an undirected graph G.

Proof. We start from the similarity between cuts in undirected graphs and $s-t$ cuts in directed graphs, and with the fact that we know how to find the latter optimally.

So given an undirected graph $G = (V, E)$, we need to transform it so that there are directed edges and there is a source and sink. We first replace every undirected edge $e = (u, v) \in E$ with two oppositely oriented directed edges, $e' = (u, v)$ and $e'' = (v, u)$, each of capacity 1. Let G' denote the resulting directed graph.

Now suppose we pick two arbitrary nodes $s, t \in V$, and find the minimum $s - t$ cut in G'. It is easy to check that if (A, B) is this minimum cut in G', then (A, B) is also a cut of minimum size in G among all those that separate s from t. But we know that the global min-cut in G must separate s from something since both sides A and B are nonempty, and s belongs to only one of them.

So we fix any $s \in V$ and compute the minimum $s-t$ cut in G' for every other node $t \in V - \{s\}$. This is $n - 1$ directed minimum-cut computations, and the best among these will be a global min-cut of G.

(13.5) The Contraction Algorithm returns a global min-cut of G with probability at least $\binom{n}{2}^{-1}$.

Proof. We focus on a global min-cut (A, B) of G and suppose it has size k; in other words, there is a set F of k edges with one end in A and the other in B. We want to give a lower bound on the probability that the Contraction Algorithm returns the cut (A, B).

Consider what could go wrong in the first step of the Contraction Algorithm: The problem would be if an edge in F were contracted. For then, a node of A and a node of B would get thrown together in the same supernode, and (A, B) could not be returned as the output of the algorithm. Conversely, if an edge not in F is contracted, then there is still a chance that (A, B) could be returned.

So what we want is an upper bound on the probability that an edge in F is contracted, and for this we need a lower bound on the size of E. Notice that if any node v had degree less than k, then the cut $(\{v\}, V - \{v\})$ would have size less than k, contradicting our assumption that (A, B) is a global min-cut. Thus every node in G has degree at least k, and so $|E| \geq \frac{1}{2}kn$. Hence the probability that an edge in F is contracted is at most $\frac{k}{\frac{1}{2}kn} = \frac{2}{n}$.

Now consider the situation after j iterations, when there are $n - j$ supernodes in the current graph G', and suppose that no edge in F has been contracted yet. Every cut of G' is a cut of G, and so there are at least k edges incident to every supernode of G'. Thus G' has at least $\frac{1}{2}k(n-j)$ edges, and so the probability that an edge of F is contracted in the next iteration $j+1$ is at most $\frac{k}{\frac{1}{2}k(n-j)} = \frac{2}{n-j}$.

The cut (A, B) will actually be returned by the algorithm if no edge of F is contracted in any of iterations $1, 2, \ldots, n-2$. If we write E_j for the event that an edge of F is not contracted in iteration j, then we have shown $\Pr[E_1] \geq 1 - \frac{2}{n}$ and $\Pr[E_{j+1}|E_1 \cap E_2 \ldots \cap E_j] \geq 1 - \frac{2}{n-j}$. We are interested in lower-bounding the quantity $\Pr[E_1 \cap E_2 \ldots \cap E_{n-2}]$, and we can check by unwinding the formula for conditional probability that this is equal to

$$\begin{aligned}
&\Pr[E_1] \cdot \Pr[E_2|E_1] \ldots \Pr[E_{j+1}|E_1 \cap E_2 \ldots \cap E_j] \ldots \Pr[E_{n-2}|E_1 \cap E_2 \ldots \cap E_{n-3}] \\
&\geq (1 - \frac{2}{n})(1 - \frac{2}{n-1}) \ldots (1 - \frac{2}{n-j}) \ldots (1 - \frac{2}{3}) \\
&= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\left(\frac{n-4}{n-2}\right) \cdots \left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\
&= \frac{2}{n(n-1)} = \binom{n}{2}^{-1}.
\end{aligned}$$

5.1.3 Complexity Analysis

The overall running time required to get a high probability of success is polynomial in n, since each run of the Contraction Algorithm takes polynomial time, and we run it a polynomial number of times. Its running time will be fairly large compared with the best network flow techniques, since we perform $\mathcal{O}(n^2)$ independent runs and each takes at least $\mathcal{O}(m)$ time. We have chosen to describe this version of the Contraction Algorithm since it is the simplest and most elegant; it has been shown that some clever optimizations to the way in which multiple runs are performed can improve the running time considerably.

Further Analysis: The Number of Global Minimum Cuts

The analysis of the Contraction Algorithm provides a surprisingly simple answer to the following question: Given an undirected graph $G = (V, E)$ on n nodes, what is the maximum number of global min-cuts it can have (as a function of n)?

For a directed flow network, it's easy to see that the number of minimum $s - t$ cuts can be exponential in n. For example, consider a directed graph with nodes $s, t, v_1, v_2, \ldots, v_n$, and unit-capacity edges (s, v_i) and (v_i, t) for each i. Then s together with any subset of $\{v_1, v_2, \ldots, v_n\}$ will constitute the source side of a minimum cut, and so there are 2^n minimum $s - t$ cuts.

But for global min-cuts in an undirected graph, the situation looks quite different. If one spends some time trying out examples, one finds that the n-node cycle has $\binom{n}{2}$ global min-cuts (obtained by cutting any two edges), and it is not clear how to construct an undirected graph with more.

Contents

1 Matrix Operations

Contents

1.1 LUP Decomposition

1.1.1 Overview of LUP Decomposition

LUP decomposition is a method used to factorize a square matrix into three components: a lower triangular matrix (L), an upper triangular matrix (U), and a permutation matrix (P). The goal is to find matrices L, U, and P such that the product of the permutation matrix P and the original matrix A is equal to the product of the lower triangular matrix L and the upper triangular matrix U, i.e., $PA = LU$.

This decomposition is particularly useful for solving systems of linear equations efficiently, as it simplifies the process by converting the original system into triangular systems that can be easily solved using forward and back substitution. The permutation matrix P allows for reordering of the rows of the matrix to improve numerical stability and efficiency in solving the system.

Overall, LUP decomposition provides a numerically stable and efficient method for solving linear systems of equations by decomposing the original matrix into lower and upper triangular matrices along with a permutation matrix.

1.1.2 Forward and Back Substitution

Forward substitution can solve the lower-triangular system (28.5) in $O(n^2)$ time, given L, P, and b. For convenience, we represent the permutation P compactly by an array $\{\pi_1, \pi_2, \ldots, \pi_n\}$. For $i = 1, 2, \ldots, n$, the entry π_i indicates that $P_{i,\pi_i} = 1$ and $P_{ij} = 0$ for $j \neq \pi_i$. Thus, PA has a π_i, j

in row i and column j, and Pb has b_{π_i} as its ith element. Since L is unit lower-triangular, we can rewrite equation as

$$\begin{aligned} y_1 &= b_{\pi_1}, \\ l_{21}y_1 + y_2 &= b_{\pi_2}, \\ l_{31}y_1 + l_{32}y_2 + y_3 &= b_{\pi_3}, \\ &\vdots \\ l_{n1}y_1 + l_{n2}y_2 + l_{n3}y_3 + \ldots + l_{n,n-1}y_{n-1} + y_n &= b_{\pi_n}. \end{aligned}$$

The first equation tells us that $y_1 = b_{\pi_1}$. Knowing the value of y_1, we can substitute it into the second equation, yielding $y_2 = b_{\pi_2} - l_{21}y_1$.

Now, we can substitute both y_1 and y_2 into the third equation, obtaining $y_3 = b_{\pi_3} - (l_{31}y_1 + l_{32}y_2)$.

In general, we substitute y_1, y_2, ..., y_{i-1} "forward" into the ith equation to solve for y_i: $y_i = b_{\pi_i} - \sum_{j=1}^{i-1} l_{ij}y_j$.

Having solved for y, we solve for x in equation (28.6) using back substitution, which is similar to forward substitution. Here, we solve the nth equation first and work backward to the first equation. Like forward substitution, this process runs in $O(n^2)$ time. Since U is upper-triangular, we can rewrite the system as

$$\begin{aligned} u_{11}x_1 + u_{12}x_2 + \ldots + u_{1,n-2}x_{n-2} + u_{1,n-1}x_{n-1} + u_{1n}x_n &= y_1, \\ u_{22}x_2 + \ldots + u_{2,n-2}x_{n-2} + u_{2,n-1}x_{n-1} + u_{2n}x_n &= y_2, \\ &\vdots \\ u_{n-2,n-2}x_{n-2} + u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n &= y_{n-2}, \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n &= y_{n-1}, \\ u_{nn}x_n &= y_n. \end{aligned}$$

Thus, we can solve for x_n, x_{n-1}, ..., x_1 successively as follows:

$$\begin{aligned} x_n &= \frac{y_n}{u_{nn}}, \\ x_{n-1} &= \frac{y_{n-1} - u_{n-1,n}x_n}{u_{n-1,n-1}}, \\ x_{n-2} &= \frac{y_{n-2} - (u_{n-2,n-1}x_{n-1} + u_{n-2,n}x_n)}{u_{n-2,n-2}}, \\ &\vdots \\ x_i &= \frac{y_i - \sum_{j=i+1}^{n} u_{ij}x_j}{u_{ii}}. \end{aligned}$$

Given P, L, U, and b, the procedure LUP-SOLVE solves for x by combining forward and back substitution. The pseudocode assumes that the dimension n appears in the attribute L : rows and that the permutation matrix P is represented by the array $\{\pi\}$.

1.1.3 Pseudocode for LUP-SOLVE

The pseudocode snippet provided is for the LUP-SOLVE procedure, which is used to solve a system of linear equations given the LUP decomposition of the coefficient matrix. Here is a step-by-step explanation of the algorithm:

Algorithm 1 LUP-SOLVE(L, U, Π, b, n)

1: Let x and y be new vectors of length n
2: **for** $i = 1$ to n **do**
3: $\quad y_i \leftarrow b_{\Pi(i)} - \sum_{j=1}^{i-1} l_{ij} y_j$
4: **end for**
5: **for** $i = n$ downto 1 **do**
6: $\quad x_i \leftarrow \left(y_i - \sum_{j=i+1}^{n} u_{ij} x_j\right) / u_{ii}$
7: **end for**
8: **return** x

1. **Input Parameters**: - L (Lower triangular matrix from LUP decomposition) - U (Upper triangular matrix from LUP decomposition) - $\{\pi\}$ (Permutation array) - b (Vector on the right-hand side of the equation)

2. **Initialization**: - Set n as the number of rows in matrix L. - Create new vectors x and y of length n to store the solutions.

3. **Forward Substitution**: - For i from 1 to n: - Calculate y_i by first computing b_{π_i} and then subtracting the sum of products of l_{ij} and y_j for j from 1 to $i - 1$.

4. **Backward Substitution**: - For i from n down to 1: - Calculate x_i by first computing y_i and then subtracting the sum of products of u_{ij} and x_j for j from $i + 1$ to n, all divided by u_{ii}.

5. **Return**: - Return the vector x containing the solutions to the system of linear equations.

The algorithm first solves for the vector y using forward substitution and then solves for the vector x using backward substitution. By combining these two steps, the algorithm efficiently computes the solution to the system of linear equations represented by the LUP decomposition.

This algorithm is based on the properties of the LUP decomposition, where the original matrix is factored into a lower triangular matrix L, an upper triangular matrix U, and a permutation

matrix P. The forward and backward substitution steps leverage the structure of these matrices to efficiently compute the solution to the system of equations.

1.1.4 Computing an LU Decomposition

We have now shown that if we can create an LUP decomposition for a nonsingular matrix A, then forward and back substitution can solve the system $Ax = b$ of linear equations. Now we show how to efficiently compute an LUP decomposition for A. We start with the case in which A is an $n \times n$ nonsingular matrix and P is absent (or, equivalently, $P = I_n$). In this case, we factor $A = LU$. We call the two matrices L and U an LU decomposition of A.

We use a process known as Gaussian elimination to create an LU decomposition. We start by subtracting multiples of the first equation from the other equations in order to remove the first variable from those equations. Then, we subtract multiples of the second equation from the third and subsequent equations so that now the first and second variables are removed from them. We continue this process until the system that remains has an upper-triangular form—in fact, it is the matrix U. The matrix L is made up of the row multipliers that cause variables to be eliminated.

Our algorithm to implement this strategy is recursive. We wish to construct an LU decomposition for an $n \times n$ nonsingular matrix A. If $n = 1$, then we are done, since we can choose $L = I_1$ and $U = A$. For $n > 1$, we break A into four parts: $A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix}$; where $w = (w_2, w_3, \ldots, w_n)^T$ is a column $(n-1)$-vector, $w^T = (a_{12}, a_{13}, \ldots, a_{1n})$ is a row $(n-1)$-vector, and A_0 is an $(n-1) \times (n-1)$ matrix. Then, using matrix algebra, we can factor A as $A = \begin{pmatrix} a_{11} & w^T \\ v & A' \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{w}{a_{11}} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A_0 - \frac{ww^T}{a_{11}} \end{pmatrix}$. The 0s in the first and second matrices of the equation are row and column $(n-1)$-vectors, respectively. The term $\frac{ww^T}{a_{11}}$, formed by taking the outer product of w and w and dividing each element of the result by a_{11}, is an $(n-1) \times (n-1)$ matrix, which conforms in size to the matrix A_0 from which it is subtracted. The resulting $(n-1) \times (n-1)$ matrix $A_0 - \frac{ww^T}{a_{11}}$ is called the Schur complement of A with respect to a_{11}.

We claim that if A is nonsingular, then the Schur complement is nonsingular, too. Why? Suppose that the Schur complement, which is $(n-1) \times (n-1)$, is singular. Then by Theorem

D.1, it has row rank strictly less than $n-1$. Because the bottom $n-1$ entries in the first column of the matrix $\begin{pmatrix} a_{11} & w^T \\ 0 & A_0 - \frac{ww^T}{a_{11}} \end{pmatrix}$ are all 0, the bottom $n-1$ rows of this matrix must have row rank strictly less than $n-1$. The row rank of the entire matrix, therefore, is strictly less than n. Applying Exercise D.2-8[1] to the equation, A has rank strictly less than n, and from Theorem D.1 we derive the contradiction that A is singular.

Because the Schur complement is nonsingular, we can now recursively find an LU decomposition for it. Let us say that $A_0 - \frac{ww^T}{a_{11}} = L_0 U_0$; where L_0 is unit lower-triangular and U_0 is upper-triangular. Then, using matrix algebra, we have $A = \begin{pmatrix} 1 & 0 \\ \frac{w}{a_{11}} & I_{n-1} \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & A_0 - \frac{ww^T}{a_{11}} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \frac{w}{a_{11}} & I_0 \end{pmatrix} \begin{pmatrix} a_{11} & w^T \\ 0 & L_0 U_0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & \\ \frac{w}{a_{11}} L_0 & a_{11} & w^T \\ 0 & U_0 & \end{pmatrix} = LU$; thereby providing our LU decomposition. (Note that because L_0 is unit lower-triangular, so is L, and because U_0 is upper-triangular, so is U).

1.1.5 Pseudocode for LU-DECOMPOSITION

Algorithm 2 LU-DECOMPOSITION(A)

```
1: n ← A.rows
2: Let L and U be new n × n matrices
3: Initialize U with 0s below the diagonal
4: Initialize L with 1s on the diagonal and 0s above the diagonal
5: for k ← 1 to n do
6:     u_kk ← a_kk
7:     for i ← k + 1 to n do
8:         l_ik ← a_ik / a_kk // a_ik holds γ_i
9:         u_ki ← a_ki // a_ki holds w_i
10:    end for
11:    for i ← k + 1 to n do
12:        for j ← k + 1 to n do
13:            a_ij ← a_ij − l_ik · u_kj
14:        end for
15:    end for
16: end for
17: return L and U
```

1.1.6 Proof of Correctness

Theorem: LUP Decomposition is a valid factorization for a nonsingular matrix A.

Proof:

Let A be an $n \times n$ nonsingular matrix. We want to show that there exist matrices L, U, and P such that $PA = LU$, where:

- L is a unit lower-triangular matrix,
- U is an upper-triangular matrix, and
- P is a permutation matrix.

We perform LUP decomposition on A to obtain $PA = LU$. Let $A = [a_{ij}]$, $L = [l_{ij}]$, $U = [u_{ij}]$, and $P = [p_{ij}]$.

The LUP decomposition algorithm ensures that:

$$\begin{aligned} PA &= LU \\ \Rightarrow P[A_0\ \gamma] &= LU \\ \Rightarrow [P_0\ P_1][A_0\ \gamma] &= [L_0\ 0][U_0\ U_1] \\ \Rightarrow P_0A_0 + P_1\gamma &= L_0U_0 \quad (1) \end{aligned}$$

Since A is nonsingular, A_0 is also nonsingular. Therefore, the Schur complement $A_0 - \gamma$ is nonsingular.

By the LUP decomposition algorithm, we have:

$$\begin{aligned} P_0A_0 + P_1\gamma &= L_0U_0 \\ \Rightarrow A_0 + P_0^{-1}P_1\gamma &= P_0^{-1}L_0U_0 \\ \Rightarrow A_0 + \tilde{\gamma} &= \tilde{L}\tilde{U} \quad (2) \end{aligned}$$

Equation (2) shows that the Schur complement $\tilde{A}_0 = A_0 + \tilde{\gamma}$ can be factorized as $\tilde{A}_0 = \tilde{L}\tilde{U}$.

Since A_0 and $\tilde{A}_0$ are nonsingular, their factorizations are unique. Therefore, the LUP decomposition of A is valid.

Hence, LUP decomposition is a valid factorization for a nonsingular matrix A.

1.1.7 Complexity Analysis

Let A be an $n \times n$ matrix for which we want to compute the LUP decomposition.

Step 1: Permutation Matrix P Constructing the permutation matrix P requires finding the maximum element in each column of A and performing row swaps. This step has a complexity of $O(n^2)$.

Step 2: LU Decomposition After obtaining the permutation matrix P, we perform LU decomposition on the permuted matrix PA. The LU decomposition involves Gaussian elimination and has a complexity of $O(n^3)$.

Overall Complexity: The total complexity of LUP decomposition is dominated by the LU decomposition step, which is $O(n^3)$. Therefore, the overall complexity of LUP decomposition for an $n \times n$ matrix A is $O(n^3)$.

2 Linear Programming

Contents

2.1 Simplex Algorithm

The Simplex algorithm is a widely used method for solving linear programming problems. It was developed by George Dantzig in 1947 and has since become one of the most popular algorithms for solving optimization problems.

Consider a linear programming problem in standard form:

$$\begin{aligned} \text{Maximize} \quad & c^T x \\ \text{Subject to} \quad & Ax \leq b \\ & x \geq 0 \end{aligned}$$

where c is the vector of coefficients in the objective function, A is the constraint matrix, b is the right-hand side vector, and x is the vector of decision variables.

The Simplex algorithm works by starting at a feasible solution and iteratively moving to adjacent feasible solutions that improve the objective function value until an optimal solution is reached. At each iteration, the algorithm pivots on a nonbasic variable to enter the basis and a basic variable to leave the basis, following the direction of steepest ascent in the objective function.

The key idea behind the Simplex algorithm is to move along the edges of the feasible region towards the optimal solution by traversing from one vertex to another, where each vertex corresponds to a basic feasible solution.

The algorithm terminates when no further improvement in the objective function value can be made, indicating that the current solution is optimal.

The efficiency of the Simplex algorithm lies in its ability to exploit the structure of the feasible region to quickly converge to the optimal solution in many practical cases.

2.1.1 Pivoting

We now formalize the procedure for pivoting. The procedure PIVOT takes as input a slack form, given by the tuple (N, B, A, b, c, γ), the index l of the leaving variable x_l, and the index e of the entering variable x_e. It returns the tuple $(\tilde{N}, \tilde{B}, \tilde{A}, \tilde{b}, \tilde{c}, \tilde{\gamma})$ describing the new slack form. (Recall again that the entries of the $m \times n$ matrices A and $\tilde{A}$ are actually the negatives of the coefficients that appear in the slack form.)

2.1.2 Pseudocode for Pivoting

Algorithm 3 PIVOT$(N, B, A, b, c, \gamma, l, e)$

// Compute the coefficients of the equation for new basic variable x_e.
let yA be a new $m \times n$ matrix
$y_{be} \leftarrow \frac{b_l}{a_{le}}$
for each $j \in N \setminus \{e\}$ **do**
    $y_{aej} \leftarrow \frac{a_{lj}}{a_{le}}$
end for
$y_{ael} \leftarrow \frac{1}{a_{le}}$
// Compute the coefficients of the remaining constraints.
for each $i \in B \setminus \{l\}$ **do**
    $y_{bi} \leftarrow b_i - a_{ie} \cdot y_{be}$
    for each $j \in N \setminus \{e\}$ **do**
        $y_{aij} \leftarrow a_{ij} - a_{ie} \cdot y_{aej}$
    end for
    $y_{ail} \leftarrow -a_{ie} \cdot y_{ael}$
end for
// Compute the objective function.
$y_\gamma \leftarrow \gamma + c_e \cdot y_{be}$
for each $j \in N \setminus \{e\}$ **do**
    $y_{cj} \leftarrow c_j - c_e \cdot y_{aej}$
end for
$y_{cl} \leftarrow -c_e \cdot y_{ael}$
// Compute new sets of basic and nonbasic variables.
$yN \leftarrow N \setminus \{e\} \cup \{l\}$
$yB \leftarrow B \setminus \{l\} \cup \{e\}$
return $(yN, yB, yA, yb, yc, y_\gamma)$

2

2.1.3 Formal Simplex Algorithm

To determine feasibility and find a feasible initial basic solution, we utilize the procedure INITIALIZE-SIMPLEX .A; b; c/ for a linear program in standard form with an $m \times n$ matrix A (a_{ij}), vectors b (b_i), and c (c_j). If the problem is infeasible, the procedure terminates with an infeasibility message; otherwise, it returns a slack form ensuring feasibility.

The SIMPLEX procedure processes linear programs in standard form, yielding an n-vector Nx (Nx_j) as the optimal solution.

2.1.4 Pseudocode for Simplex Algorithm

Algorithm 4 SIMPLEX Algorithm

```
1:  procedure SIMPLEX(A, b, c)
2:      (N, B, A, b, c, γ) ← INITIALIZE-SIMPLEX(A, b, c)
3:      Let λ be a new vector of length m
4:      while some index j ∈ N has c_j > 0 do
5:          Choose an index e ∈ N for which c_e > 0
6:          for each index i ∈ B do
7:              if a_ie > 0 then
8:                  λ_i ← b_i / a_ie
9:              else
10:                 λ_i ← 1
11:             end if
12:         end for
13:         Choose an index l ∈ B that minimizes λ_l
14:         if λ_l = 1 then
15:             return "unbounded"
16:         else
17:             (N, B, A, b, c, γ) ← PIVOT(N, B, A, b, c, γ, l, e)
18:         end if
19:     end while
20:     for i = 1 to n do
21:         if i ∈ B then
22:             Nx_i ← b_i
23:         else
24:             Nx_i ← 0
25:         end if
26:     end for
27:     return Nx_1, Nx_2, ..., Nx_n
28: end procedure
```

2.1.5 Correctness Proof of Simplex Algorithm

Lemma 29.12: If a linear program L has no feasible solution, then INITIALIZE-SIMPLEX returns "infeasible." Otherwise, it returns a valid slack form for which the basic solution is feasible.

Proof: First suppose that the linear program L has no feasible solution. Then by Lemma 29.11, the optimal objective value of L_{aux}, defined in (29.106)-(29.108), is nonzero, and by the nonnegativity constraint on x_0, the optimal objective value must be negative. Furthermore, this objective value must be finite, since setting $x_i = 0$, for $i = 1, 2, \ldots, n$, and $x_0 = \min_{i=1}^{m}\{b_i\}$ is feasible, and this solution has objective value $-\min_{i=1}^{m}\{b_i\}$. Therefore, line 10 of INITIALIZE-SIMPLEX finds a solution with a nonpositive objective value. Let N_x be the basic solution associated with the final slack form. We cannot have $N_x^0 = 0$, because then L_{aux} would have objective value 0, which contradicts that the objective value is negative. Thus the test in line 11 results in line 16 returning "infeasible."

Suppose now that the linear program L does have a feasible solution. From Exercise 29.3-4, we know that if $b_i \geq 0$ for $i = 1, 2, \ldots, m$, then the basic solution associated with the initial slack form is feasible. In this case, lines 2-3 return the slack form associated with the input. (Converting the standard form to slack form is easy, since A, b, and c are the same in both.)

In the remainder of the proof, we handle the case in which the linear program is feasible but we do not return in line 3. We argue that in this case, lines 4-10 find a feasible solution to L_{aux} with objective value 0. First, by lines 1-2, we must have $b_k < 0$; and $b_k \leq b_i$ for each $i \in B$. In line 8, we perform one pivot operation in which the leaving variable x_l (recall that $l = n+k$, so that $b_l < 0$) is the left-hand side of the equation with minimum b_i, and the entering variable is x_0, the extra added variable. We now show that after this pivot, all entries of b are nonnegative, and hence the basic solution to L_{aux} is feasible. Letting N_x be the basic solution after the call to PIVOT, and letting y_b and y_B be values returned by PIVOT, Lemma 29.1 implies that $N_x^i = \begin{cases} b_i - a_i^e y_b^e & \text{if } i \in y_B - \{e\} \\ b_l = a_l^e & \text{if } i = e \end{cases}$.

The call to PIVOT in line 8 has $e = 0$. If we rewrite inequalities (29.107) to include coefficients a_i^0, $\sum_{j=0}^{n} a_{ij}x_j \leq b_i$ for $i = 1, 2, \ldots, m$, then $a_i^0 = a_i^e = -1$ for each $i \in B$. Since $l \in B$, we also have that $a_l^e = -1$. Thus, $b_l/a_l^e > 0$, and so $N_x^e > 0$. For the remaining basic variables, we have $N_x^i = b_i - a_i^e y_b^e$ (by equation (29.113)) $= b_i - a_i^e \cdot b_l/a_l^e$ (by line 3 of PIVOT) $= b_i - b_l$ (by equation (29.115) and $a_l^e = -1$) ≥ 0 (by inequality (29.112)), which implies that each basic variable is now nonnegative. Hence the basic solution after the call to PIVOT in line 8 is feasible. We next execute line 10, which solves L_{aux}. Since we have assumed that L has a feasible solution, Lemma 29.11 implies that L_{aux} has an optimal solution with objective value 0. Since all the slack forms are equivalent, the final basic solution to L_{aux} must have $N_x^0 = 0$, and after removing x_0 from the linear program, we obtain a slack form that is feasible for L. Line 15 then returns this slack form.

2.1.6 Complexity Analysis

Let n be the number of variables and m be the number of constraints in the linear programming problem.

The worst-case complexity of the Simplex Algorithm is exponential in the worst case, specifically $O(2^n)$, as there exist instances where the algorithm may take an exponential number of steps to converge.

However, in practice, the Simplex Algorithm tends to perform well and solve most linear programming problems efficiently. The average-case complexity of the Simplex Algorithm is polynomial, making it a practical choice for many real-world optimization problems.

Therefore, the complexity of the Simplex Algorithm can be summarized as follows:

- Worst-case complexity: $O(2^n)$
- Average-case complexity: Polynomial

2.2 Duality

Linear programming duality is a fundamental concept in optimization theory that establishes a strong relationship between primal and dual optimization problems. Let us consider a primal linear program in standard form:

$$\text{Maximize} \quad c^T x \text{ Subject to} \qquad Ax \leq b \; x \geq 0$$

where c is the vector of coefficients in the objective function, A is the constraint matrix, b is the right-hand side vector, and x is the vector of decision variables.

The dual linear program associated with the primal problem is formulated as:

$$\text{Minimize} \quad b^T y \text{ Subject to} \qquad A^T y \geq c \; y \geq 0$$

where y is the vector of dual variables corresponding to the constraints in the primal problem.

The duality theory establishes a strong relationship between the optimal objective values of the primal and dual problems. Specifically, if x $*$ and y $*$ are optimal solutions to the primal and dual problems respectively, then the duality theorem states that:

$$c^T x^* \leq b^T y^*$$

This inequality provides a valuable insight into the optimality of solutions to linear programming problems and forms the basis for duality theory in optimization.

In the subsequent sections, we will explore the properties of duality, the implications of the duality theorem, and how duality theory enhances our understanding and solving of linear programming problems.

2.2.1 Pseudocode

Algorithm 5 Linear Programming Duality

```
Input: Primal linear program in standard form
Output: Dual linear program
procedure FormulateDual
    Let the primal linear program be:
    Maximize c T x
    Subject to Ax b, x 0
    Define the dual linear program as:
    Minimize b^T y
    Subject to A^T y ≥ c, y ≥ 0
end procedure
```

2.2.2 Proof of Correctness

Let the primal linear program be denoted by P and its corresponding dual by D. If both P and D are feasible and bounded, then for optimal solutions x and y, we have $c^T x = b^T y$.

Proof: Let $\theta = b^T y$ be the optimal value of the dual linear program D. Consider an augmented set of primal constraints where we add a constraint that the objective value is at least θ. This augmented primal can be written as: $Ax \leq b, \quad c^T x \geq \theta$

Multiplying the inequality $c^T x \geq \theta$ by -1 and rewriting the constraints, we get: $\sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad -\sum_{j=1}^{n} c_j x_j \leq -\theta$

Here, A is an $m \times n$ matrix, x is an n-vector, and b is an m-vector.

Therefore, for optimal solutions x and y, we have shown that $c^T x = b^T y$, proving the correctness of Duality.

2.2.3 Complexity Analysis

The complexity of Duality in linear programming is primarily determined by the complexity of solving the primal and dual linear programs. Let n be the number of variables and m be the number of constraints in the linear programming problem.

The complexity of Duality is typically polynomial, as it involves solving two linear programs (primal and dual) which can be done efficiently using algorithms like the Simplex Algorithm or Interior-Point Methods. The complexity of solving each linear program is usually polynomial in the input size.

Therefore, the overall complexity of Duality in linear programming can be considered polynomial, making it a computationally efficient concept for optimization problems.

3 Number-Theoretic Algorithms

Contents

3.1 Euclid's algorithm

Given two non-negative integers a and b, where $a \geq b$, Euclid's algorithm proceeds as follows:

- If $b = 0$, the algorithm terminates, and the GCD is a.
- Otherwise, the algorithm recursively applies the same process with b and the remainder of a divided by b, denoted as $a \mod b$.

The recursive nature of Euclid's algorithm ensures that it will eventually terminate since the second argument in each recursive call strictly decreases and is always non-negative. The correctness of the algorithm follows from the fact that if the algorithm returns a in the base case, then $b = 0$, and the GCD is indeed a.

Euclid's algorithm is efficient and widely used due to its simplicity and effectiveness in computing the GCD of two integers. It forms the basis for many other number-theoretic algorithms and is a fundamental tool in various mathematical applications, including cryptography, number theory, and computer science.

Algorithm 6 Euclid's Algorithm

```
1: function EUCLID(a, b)
2:     if b = 0 then
3:         return a
4:     else
5:         return EUCLID(b, a mod b)
6:     end if
7: end function
```

3.1.1 Pseudocode

3.1.2 Proof of Correctness

Let a and b be two non-negative integers with $a \geq b$. We want to show that Euclid's Algorithm correctly computes the greatest common divisor (GCD) of a and b.

Base Case: If $b = 0$, the algorithm terminates and returns a, which is the GCD of a and b.

Inductive Step: Suppose the algorithm correctly computes the GCD of b and $a \mod b$ for some a and b with $b \neq 0$. Let d be the GCD of a and b. By the division algorithm, we can write $a = qb + r$, where q is the quotient and r is the remainder.

Since d divides both a and b, it also divides $r = a - qb$. Therefore, d divides b and r, implying that d is the GCD of b and r.

By the inductive hypothesis, the algorithm correctly computes the GCD of b and r, which is the same as the GCD of a and b.

Hence, by induction, Euclid's Algorithm correctly computes the GCD of any two non-negative integers a and b.

3.1.3 Complexity Analysis

Let a and b be non-negative integers where we want to compute the greatest common divisor $\gcd(a, b)$ using Euclid's Algorithm.

Time Complexity:

The worst-case time complexity of Euclid's Algorithm can be analyzed in terms of the number of recursive calls made.

Assuming $a > b \geq 0$, the algorithm makes at most $\log_\phi a$ recursive calls, where ϕ is the golden ratio.

Therefore, the time complexity of Euclid's Algorithm is $O(\log_\phi a)$, where a is the larger of the two input integers.

3.2 Chinese Remainder Theorem

The Chinese Remainder Theorem (CRT) is a fundamental result in number theory that provides a solution to a system of simultaneous congruences. It allows us to reconstruct an integer from its remainders modulo a set of pairwise relatively prime moduli.

Consider a system of congruences:
$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \vdots \\ x \equiv a_n \pmod{m_n} \end{cases}$$

where $m_1, m_2, \ldots, m_n$ are pairwise relatively prime integers. The Chinese Remainder Theorem states that this system of congruences has a unique solution modulo $M = m_1 \cdot m_2 \cdots m_n$.

The theorem has applications in various areas of mathematics and computer science, including cryptography, number theory, and algorithm design. It provides a powerful tool for solving problems involving modular arithmetic and finding solutions to systems of congruences.

3.2.1 Pseudocode

Algorithm 7 Chinese Remainder Theorem

```
function ChineseRemainder(a_1, m_1, a_2, m_2, ..., a_n, m_n)
    M ← m_1 · m_2 ··· m_n
    M_i ← M/m_i for i = 1, 2, ..., n
    Compute y_i such that M_i y_i ≡ 1 (mod m_i) for each i
    x ← 0
    for i = 1 to n do
        x ← x + a_i M_i y_i
    end for
    return x mod M
end function
```

3.2.2 Proof of Correctness

We first prove that there exists a solution in $\mathbb{Z}$; we will do this algorithmically.

Consider the first two congruences in the system:

$$x \equiv a_1 \pmod{m_1},$$
$$x \equiv a_2 \pmod{m_2}.$$

Since m_1 and m_2 are relatively prime, Bezout's Lemma implies that there exist integers s_1 and t_1 such that $s_1 m_1 + t_1 m_2 = 1$.

We find these integers, perhaps using a technique from Section 1.10. Once we have, we construct a partial solution by $x_2 = a_1 t_1 m_2 + a_2 s_1 m_1$.

Note here that we have attached the summands in equation to the right-hand sides of the congruences, but in the "opposite" order. We use the subscript 2 and call x_2 a partial solution because it satisfies the first two congruences we began with; and a similar argument shows that x_2.

Our next step is to record the information about our partial solution in its own congruence and pair it with the next unused congruence from our system:

$$x \equiv x_2 \pmod{m_1 m_2},$$
$$x \equiv a_3 \pmod{m_3}.$$

We solve this pair of congruences the same way we did the first pair; since m_1 and m_2 can have no common prime factors, Bezout's Lemma implies the existence of integers s_2 and t_2 such that $s_2 m_1 m_2 + t_2 m_3 = 1$, and our next partial solution is $x_3 = x_2 t_2 m_3 + a_3 s_2 m_1 m_2$.

It's possible to check that x_3 satisfies the congruences modulo m_1, m_2, and m_3. We once again move on to the next pair of congruences, using our partial solution and the next unused congruence:

$$x \equiv x_3 \pmod{m_1 m_2 m_3},$$
$$x \equiv a_4 \pmod{m_4}.$$

We continue in this way until we reach the partial solution x_k, which will be a solution to all of the original congruences. Thus the system has at least one solution.

Uniqueness of Solution:

Suppose x' is another solution to the system. Then, for each i, we have $x \equiv a_i \equiv x' \pmod{n_i}$. This implies n_i divides $x - x'$ for all i. Since n_i are pairwise relatively prime, their product n divides $x - x'$. Hence, $x \equiv x' \pmod{n}$.

Therefore, the system of congruences has a unique solution modulo n.

3.2.3 Complexity Analysis

Let $n = n_1 \cdot n_2 \cdots n_k$, where n_i are pairwise relatively prime integers. We analyze the complexity of the Chinese Remainder Theorem algorithm.

Step 1: Computing M Computing $M = n_1 \cdot n_2 \cdots n_k$ requires $O(k)$ multiplications.

Step 2: Computing M_i For each i, computing $M_i = M/n_i$ requires $O(k)$ divisions.

Step 3: Computing y_i For each i, computing y_i such that $M_i y_i \equiv 1 \pmod{n_i}$ can be done using the Extended Euclidean Algorithm, which has a time complexity of $O(\log n_i)^2$.

Step 4: Combining Solutions Combining the solutions $a_i M_i y_i$ for $i = 1, 2, \ldots, k$ requires $O(k)$ additions.

Therefore, the overall time complexity of the Chinese Remainder Theorem algorithm is dominated by the computation of y_i, resulting in a total time complexity of $O(k \cdot (\log n)^2)$.

3.3 Repeated Squaring Algorithm

The Repeated Squaring Algorithm is a method used for efficiently computing exponentiation modulo a number. Given integers a, b, and n, the algorithm calculates $a^b \mod n$.

The algorithm works by repeatedly squaring the base a and reducing the exponent b until b becomes zero. At each step, if the current bit of the exponent b is 1, the result is multiplied by the current base value and then reduced modulo n. If the current bit of b is 0, the base value is squared modulo n.

The Repeated Squaring Algorithm is particularly useful in number theory and cryptography, where modular exponentiation is a common operation. By reducing the number of multiplications and modular operations required, the algorithm provides a more efficient way to compute large exponentiations modulo a given number.

The time complexity of the Repeated Squaring Algorithm is logarithmic in the size of the exponent b, making it a preferred method for modular exponentiation in scenarios where efficiency is crucial.

3.3.1 Pseudocode

Algorithm 8 Repeated Squaring Algorithm

```
1: function RepeatedSquaring(a, b, n)
2:     result ← 1
3:     base ← a mod n
4:     while b > 0 do
5:         if b mod 2 = 1 then
6:             result ← (result · base) mod n
7:         end if
8:         base ← (base · base) mod n
9:         b ← b ÷ 2
10:    end while
11:    return result
12: end function
```

3.3.2 Proof of Correctness

Let a, b, and n be integers where we want to compute $a^b \mod n$ using the Repeated Squaring Algorithm.

Base Case: For $b = 0$, the algorithm correctly returns 1 as $a^0 \equiv 1 \pmod{n}$.

Inductive Step: Assume the algorithm correctly computes $a^k \mod n$ for some $k \geq 0$. We will show that it also computes $a^{2k} \mod n$ correctly.

When the algorithm computes $a^{2k} \mod n$, it squares the result of $a^k \mod n$ and reduces it modulo n. By the inductive hypothesis, we know that $a^k \equiv x \pmod{n}$ for some x. Therefore, $(a^k)^2 \equiv x^2 \pmod{n}$.

Since $(a^k)^2 \equiv a^{2k} \pmod{n}$, the algorithm correctly computes $a^{2k} \mod n$.

Hence, by induction, the Repeated Squaring Algorithm correctly computes $a^b \mod n$ for any non-negative integer b.

3.3.3 Complexity Analysis

Let a, b, and n be integers where we want to compute $a^b \mod n$ using the Repeated Squaring Algorithm.

The Repeated Squaring Algorithm reduces the exponent b by half in each iteration. Therefore, the number of iterations required to compute $a^b \mod n$ is proportional to the number of bits in the binary representation of b.

Let k be the number of bits in the binary representation of b. The algorithm performs at most $2k$ modular multiplications and at most k modular squarings.

Hence, the time complexity of the Repeated Squaring Algorithm is $O(k)$, where k is the number of bits in the binary representation of the exponent b.

Contents

1 Baker's Technique

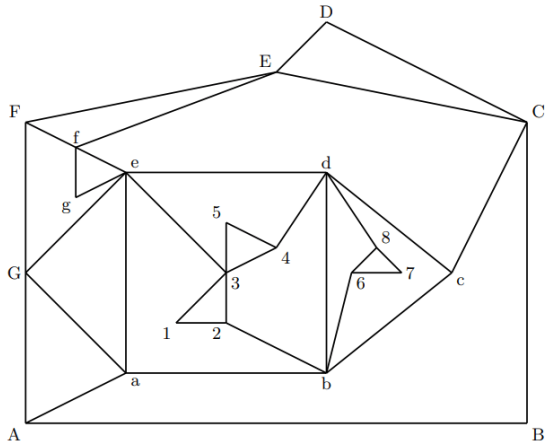
Contents

References to B. S. Baker, "Approximation algorithms for NP-complete problems on planar graphs," 24th Annual Symposium on Foundations of Computer Science (sfcs 1983), pp. 265–273, 1983 [1] is included in this chapter. Please refer the original paper for additional information.

1.1 Introduction

Approximation algorithms for NP-complete problems on planar graphs, as outlined by Brenda S. Baker, offer a mathematical framework for efficiently solving challenging computational tasks [1]. The paper focuses on Baker's approximation scheme for planar graphs. It delves into various strategies to tackle NP-complete problems on graphs, emphasizing the importance of exploring different approaches due to the unknown complexity of such problems. The key points covered in the paper include:

- **Planar Graphs**: Limiting the problem to a subclass like planar graphs can lead to more efficient algorithms due to the inherent structures present in these graphs.
- **Approximation Algorithms**: These algorithms provide a way to approximate optimal solutions with a polynomial runtime and a bound on the approximation ratio, offering flexibility based on the desired accuracy.



- **Fixed Parameter Tractability**: By introducing an additional parameter k, the complexity of NP-hard problems can be refined to achieve good running times for instances with low k, particularly beneficial for graphs where complexity increases with graph complexity.
- **Dynamic Programming**: A popular paradigm for solving problems by breaking them into subproblems and recursively merging solutions to compute the optimal solution for the main instance.
- **Purpose of the Paper**: The paper aims to apply the principles discussed to solve NP-complete problems on planar graphs, building on approximation algorithms for such problems.

The paper focuses on the Maximum Independent Set problem, adapting techniques to solve outerplanar graphs initially and then generalizing the algorithm to handle all planar graphs. The main innovation lies in decomposing planar graphs into slices to facilitate dynamic programming, with the algorithm achieving exact solutions for k-outerplanar graphs with a linear runtime in the number of nodes but exponential in k. Additionally, the algorithm is incorporated into a polynomial time approximation scheme for solving the Maximum Independent Set problem with an approximation ratio of $\frac{k}{k+1}$ in $O(8^k kn)$ time.

1.2 An Approximation Algorithm for Planar Graphs

We aim to construct a polynomial time approximation scheme for the Maximum Independent Set problem on planar graphs using the concept of k-outerplanarity. The algorithm, designed for a fixed positive integer k, seeks to solve the Maximum Independent Set with an approximation ratio

of $\frac{k}{k+1}$ and linear complexity in terms of the number of nodes n.

The algorithm focuses on solving the Maximum Independent Set on disjoint subgraphs that are k-outerplanar, while disregarding certain nodes to merge the subgraphs without violating the conditions of the Maximum Independent Set. The formalization of this concept is captured in Theorem 1:

Theorem 1 [Baker, 1994]: For a positive integer k, given a k-outerplanar embedding of a k-outerplanar graph G, an optimal solution for the Maximum Independent Set can be obtained in time $O(8kn)$, where n is the number of nodes.

Given a positive integer k and a planar graph G, the algorithm operates as follows:

1. Generate a planar embedding of G and compute the level of every node.
2. For each i, $0 \leq i \leq k$, do the following:
   (a) Remove nodes where the node's level mod $(k+1)$ equals i, splitting the graph into components with a k-outerplanar embedding.
   (b) Use Theorem 1 to solve the Maximum Independent Set on all components and take the union of the solutions.
3. Select the best solution for all i as the solution for the entire graph.

The algorithm achieves an approximation ratio of $\frac{k}{k+1}$, as formalized in Theorem 2:

Theorem 2 [Baker, 1994]: For fixed k, there exists an $O(8^k kn)$-time algorithm for the Maximum Independent Set problem, providing a solution of size at least $\frac{k}{k+1}$ optimal for general planar graphs. By choosing $k = dc \log\log n$, where c is a constant, the algorithm runs in time $O(n(\log n)^{3c} \log\log n)$ and achieves a solution of size at least $\frac{c \log\log n}{1+c \log\log n}$ optimal.

1.3 An Exact Algorithm for Outerplanar Graphs

Consider the special case of outerplanar graphs. To solve k-outerplanar graphs, analyze the algorithm in depth while keeping adaptability in mind. The algorithm is based on dynamic programming, recursively merging solutions for subgraphs of G to construct an optimal solution for the whole graph. Efforts are made to choose subgraphs efficiently to ensure constant time for each merger.

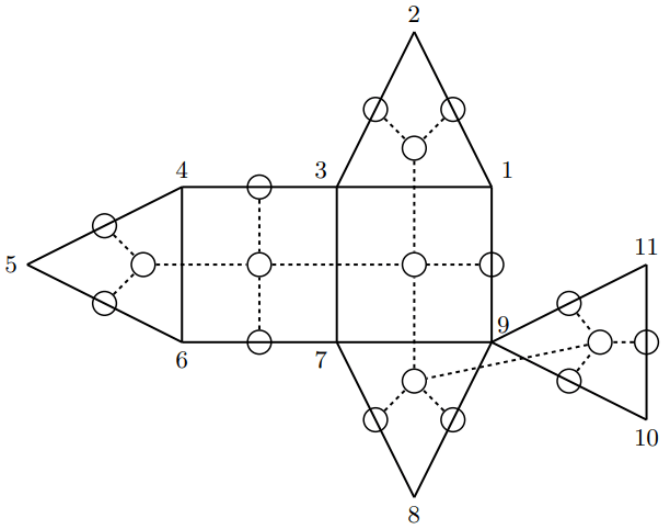


Figure 2. A tree for an outerplanar graph [Baker, 1994].

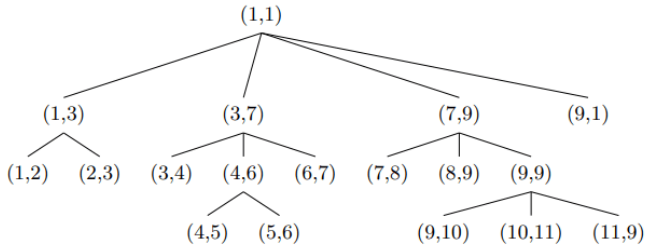


Figure 3. The tree from Figure 2 ordered and labelled [Baker, 1994].

1.3.1 Constructing the Tree G Representing G

An easy way to merge optimal solutions for two graphs is needed. An optimal solution for a subgraph may not be a subset of the solution for the merged subgraph or the entire graph due to additional constraints on nodes adjacent to nodes outside the subgraph. Boundary nodes induce the optimal assignment for other nodes. Any outerplanar graph G can be decomposed such that each subgraph has two boundary nodes. This decomposition is visualized by a tree G.

1. Remove each bridge, an edge whose removal disconnects the graph, by adding a second edge between the same nodes, making it a face.

2. Construct a graph vertex for every interior face and every exterior edge.

3. Draw edges between every edge vertex and the vertex of the face the edge is contained in, and between every two vertices of faces that share an edge.

4. Identify cutpoints, nodes where at least two faces meet without sharing an edge, and draw edges between vertices of faces containing the cutpoint until G is connected.

See Figure 2 for an example of a graph and its tree. Note that node 9 is a cutpoint.

1.3.2 Labelling G to represent a walk around G

- Choose all edge vertices as leaves of G.
- Let a face vertex be the root, with any adjacent vertex as the first child.
- Label vertices with two-tuples of nodes from G, representing subgraph boundaries.
- Subgraph nodes are on a counterclockwise walk from the first to the second boundary node.
- Inner vertices represent shortcuts on the walk.
- Label each edge vertex with its endpoints and each face vertex with left and right child labels.

1.3.3 The dynamic program computing the optimal solution

- Use dynamic programming on tree G to construct a table with four entries for each vertex, encoding combinations for two boundary nodes.
- Entries represent the size of the optimal solution for the subgraph with the given boundary nodes.
- For a vertex v:
  - If v is a level 1 leaf with label (x,y), return a table representing (x,y).
  - Otherwise, let T=table(u), where u is the leftmost child of v.
  - For each other child c of v from left to right, merge T with table(c).
  - Return adjust(T).
- Merge two tables T_1 and T_2 bound by (x, y) and (y, z) to construct a table for (x, z) by considering all assignments of x and z.
- Generalize the merge formula to maximize over all possible assignments of y to generate the entry $T(x, z)$ for boundaries x, z.

Algorithm 1 Table Algorithm

```
1: function TABLE(v)
2:     if v is a level 1 leaf with label (x, y) then
3:         return a table representing (x, y)
4:     else
5:         T ← table(u), where u is the leftmost child of v
6:         for each other child c of v from left to right do
7:             T ← merge(T, table(c))
8:         end for
9:         return adjust(T)
10:    end if
11: end function
```

- Adjust entries to handle adjacent or same boundary nodes, ensuring correctness.
- The size of the maximum independent set of G is the larger value of entries for $(0,0)$ and $(1,1)$ in the table obtained by calling table on the root of G.

1.4 On the Notion of Slices

1.4.1 The basic idea

Let us examine the properties of k-outerplanar graphs that can be useful in defining a new type of subgraph. These graphs are planar, so selecting a path between two exterior nodes as a boundary will always divide the graph into two parts. To maintain low complexity, we aim to choose two nodes such that the length of the path between them is minimized, ideally bounded. This mirrors our approach with outerplanar graphs, where finding exterior nodes with short paths was straightforward. Now, we must work within the broader constraint of a k-outerplanar graph.

Consider a node of level i. While a path from this node to an exterior node may be arbitrarily long, we can identify $i-1$ nodes, each from levels 1 to i, such that a path using only those nodes preserves planarity. This necessitates additional edges, utilized in constructing slices but not in computing the optimal solution. By combining the paths of two nodes of level i that share an edge or are identical, we form a boundary of $2i$ nodes. The left boundary comprises the path from an exterior edge to the first node, and the right boundary consists of the path from an exterior node to the second node. Consequently, we now view boundaries as two vectors of nodes rather than individual nodes.

To illustrate this concept further, consider the analogy:

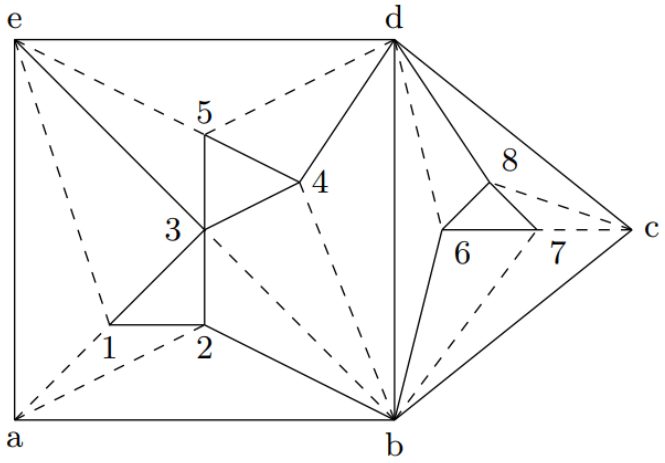


Figure 4. Triangulating regions between levels 2 and 3 for the graph of Figure 1. Edges added in the triangulation are shown as dashed lines [Baker, 1994].

Think of the entire graph as a pie, and you aim to slice out a portion. Initially, make a cut towards the center, not exceeding the pie's radius. Subsequently, make a second cut ending precisely where the first cut concluded, again within the pie's radius, resulting in a slice. This parallels our approach, with the left boundary akin to the first cut and the right boundary akin to the second cut.

1.4.2 Constructing Trees for k-Outerplanar Graphs

We will leverage our tree concept to depict the original graph's structure. Each separated level i component is outerplanar, enabling us to construct a tree that represents a counterclockwise traversal along the component's exterior edges. This process yields distinct trees for each component, constructed based on the enclosing component's trees. The merging of subgraphs is ensured through the construction of slices.

The key difference lies in selecting the root and leftmost child for the level i component trees, contingent on the enclosing component's root. Consider a level i component C enclosed by a level $i-1$ face f labeled (x, y). Note that the labels are assigned based on the exterior edge walk of the component, comprising two nodes rather than node vectors like the boundaries, although denoted as (x, y) for both. By simultaneously scanning their nodes, a triangulation between x and y is constructed in linear time. Refer to Figure 4 for an illustration of such a triangulation. The root (z, z) for C is determined by (x, y). If $x = y$, then z can be any node adjacent to x; otherwise, it is the node adjacent to both x and y in the triangulation. Subsequently, the leftmost child is the first

edge counterclockwise from (z, x), uniquely defining the tree's structure as previously described.

1.5 Formal Definition of Slices

1.5.1 A first approach

Let v be a tree vertex labeled (x, y). We define the slices for each case as follows:

1. If v represents a level i face with no enclosed nodes, $i \geq 1$, its slice is the union of the slices of its children, plus (x, y).

2. If v represents a level i face enclosing a level $i+1$ component C, its slice is that of the root of the tree for C plus (x, y). (However, the boundaries only run from level i to level 1 instead of from level $i + 1$ to 1.)

3. If v represents a level 1 edge, its slice is the subgraph consisting of (x, y).

4. If v represents a level i edge, $i > 1$, then its slice includes (x, y), edges from x and y to level $i - 1$ nodes, and the slices computed recursively for appropriate level $i - 1$ trees. Here, 'appropriate' is determined by slice boundaries placed along edges in a triangulation of the region between level $i - 1$ and i [Baker, 1994].

These definitions ensure that the slices contain nodes from all levels up to the level of the slice, facilitating the merging of leaves and maintaining the integrity of the boundaries.

1.5.2 Computing slices across levels

We can assume by induction that using the left boundary of some level $i - 1$ vertex and the right boundary of some other level i vertex, which is to the right of the first vertex considering the tree from the level i component or simply the same vertex, yields a correct boundary if the hole between their boundaries is properly filled. Therefore, we will produce a function assigning two level $i - 1$ vertices to each level i vertex which fills said hole in the boundary. The triangulation used for defining the trees will be recycled here, as it gives us an idea which pairs of nodes would be suitable partners in a boundary.

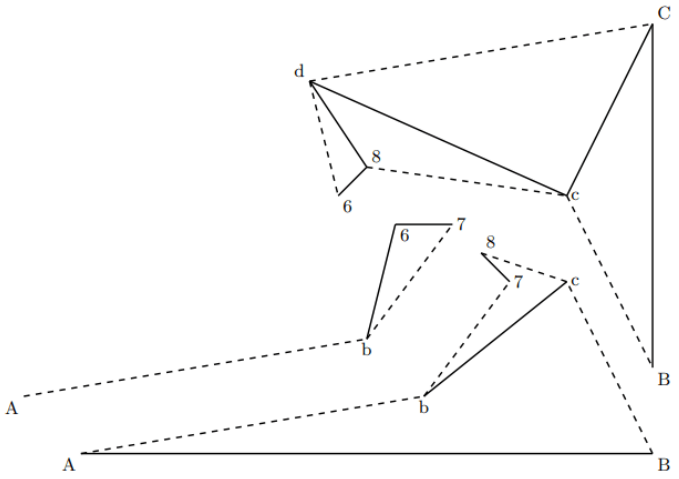


Figure 5. Three slices for the graph of Figure 1. Missing edges between boundary nodes are shown as dashed lines [Baker, 1994].

For this function, we first need to define dividing points. Let C be a level i component enclosed by a level $i-1$ face f whose tree vertex is labeled (x, y). Let $TRI(C, f)$ be the triangulation of the region between C and f already constructed in defining the trees. For any pair of successive edges (x_1, x_2), (x_2, x_3) in a counterclockwise walk around the exterior edges of C, there is at least one node y of f that is adjacent to x_2 in $TRI(C, f)$ such that the edges (x_2, x_1), (x_2, y), (x_2, x_3) occur in counterclockwise order around x_2. Call such a node a dividing point for (x_1, x_2) and (x_2, x_3) [Baker, 1994].

Basically, every node of the face adjacent to x_2 is a dividing point for the two exterior edges including x_2, except when there is a cutpoint. A cutpoint has more than two exterior edges, so we can't simply assign nodes to their dividing points for the boundary and need the more complicated definition above. Again, see Figure 4 for examples of dividing points. a and b are dividing points for the edges $(1, 2)$ and $(2, 3)$. e is a cutpoint for the edges $(5, 3)$ and $(3, 1)$, but b is not as 3 is a cutpoint.

We will now define the functions assigning boundaries to all level i vertices, which we will call LB and RB. These functions will map the vertices of C on the numbers 1 to r where r is the number of children of the tree vertex corresponding to f. The definition is as follows:

1. Let the leaves of C be $v_1, v_2, \ldots, v_t$ from left to right, and let v_j have label (x_j, x_{j+1}), for $1 \leq j \leq t$. Let the children of vertex f be $z_1, z_2, \ldots, z_r$ from left to right, where z_j has labeled (y_j, y_{j+1}) for $1 \leq j \leq r$. Define $LB(v_1) = 1$ and $RB(v_t) = r + 1$. For $1 < j \leq t$, define $LB(v_j) = q$ if q is the least $p \geq LB(v_{j-1})$ for which y_p is a dividing point for (x_{j-1}, x_j) and

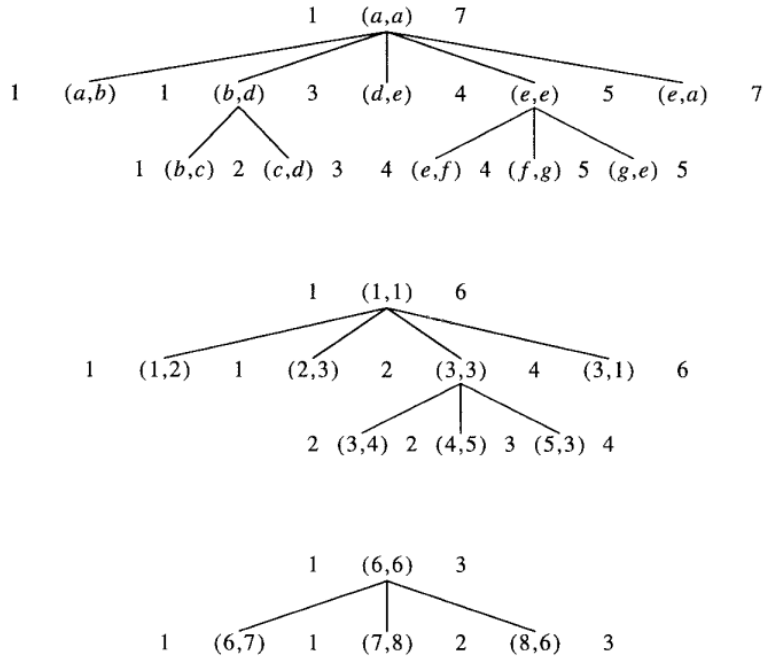


FIG. 11. Values of LB and RB for the level-2 and level-3 trees of Figure 9.

x_j, x_{j+1}. For $1 \leq j < t$, define $RB(v_j) = LB(v_{j+1})$.

2. If v is a face vertex of C, with leftmost child c_L and rightmost child c_R, define $LB(v) = LB(c_L)$ and $RB(v) = RB(c_R)$ [Baker, 1994].

Remember that we want to do a counterclockwise walk around the edges of f, but right now we do not want to consider the edge corresponding to the face's label as its slice has already been taken care of, so we make sure to start exactly to the right of it and end exactly to the left by setting the values for $LB(v_1)$ and $RB(v_t)$ accordingly.

PROPOSITION 1. For any face vertex v, the left boundary of v's slice is the same as the left boundary of the slice of v's leftmost child, and the right boundary of v's slice is the same as the right boundary of the slice of v's rightmost child.

PROOF. By rule B2, the first node in the boundary of a vertex v is the first node in v's label and the remainder is determined by $LB(u)$. The labeling of trees causes the first node in the label of v to be the same as the first node in the label of its leftmost child C_L. By rule F2, $LB(v)$ is defined to be the same as $LB(C_L)$. A similar analysis applies to right boundaries.

This gives us the formal definition of slices: Let v be a level i vertex, $i \geq 1$, with label (x, y). Again, if a vertex u has s children, define the left boundary of the $(s+1)$th child to be the same as the right boundary of the sth child.

1. If v is a face vertex, and face(v) encloses no level $i+1$ nodes, then slice(v) is the union of the slices of the children of v, together with (x, y) if (x, y) is an edge (i.e., $x \neq y$).

2. If v is a face vertex and face(v) encloses a level $i+1$ component, then slice(v) is the subgraph containing slice(root(C)) plus (x, y) if (x, y) is an edge.

3. If v is a level 1 leaf, then slice(v) is the subgraph consisting of (x, y).

4. Suppose v is a level i leaf, $i > 1$. Suppose the enclosing face is f, and vertex(f) has children u_j, $1 \leq j \leq t$, where u_j has label (z_j, z_{j+1}). If LB(v) $\neq$ RB(v), then slice(v) is the subgraph containing (x, y), any edges from x or y to z_j, for LB(v) $\leq j \leq$ RB(v), and slice(u_j), for LB(v) $\leq j \leq$ RB(v). If LB(v) = RB(v) = r, then slice(v) is the subgraph containing (x, y), any edges from x or y to z_r, the left boundary of slice(u_r), and any edges between boundary nodes of successive levels [Baker, 1994].

This definition incorporates all of our thoughts collected before. See Figure 5 for the slices of the level 3 vertices $(6, 7)$, $(7, 8)$, $(8, 6)$ in Figure 1. As you can see, the boundaries align so that the slices can be merged easily. For example, the slice of the vertex labelled (b, d), which represents the face enclosing the level 3 component, can now easily be computed by merging on the right boundary of $(6, 7)$, which is 7, b, A and equal to the left boundary of $(7, 8)$, and merging the resulting slice with the slice for $(8, 6)$, then finally adding the edge (b, d).

PROPOSITION 2. The slice for any vertex u of a tree T includes the slices of all its descendants in T plus the slices of all vertices for components enclosed by faces corresponding to descendants of u.

PROOF. The proof is by double induction starting with level k. For level k, rule $S1$ applies to each face vertex, and consequently, each level k face vertex includes the slices of its descendants (no components are enclosed by level k faces).

Assume that the result holds for level j, and consider a vertex u in a level $j-1$ tree. If u is a leaf, the statement is vacuously true. Assume that the result holds for all descendants of u. Then, either rule $S1$ applies, implying that the slice for u is the union of the slices of u's children, or rule $S2$ applies, implying that the slice for u includes the slice of the root of C, where C is the component enclosed by face(u).

In the latter case, by the induction hypothesis, the slice of the root of C includes the slices of the leaves of C, as well as the slices of vertices for components enclosed by faces of C. By the

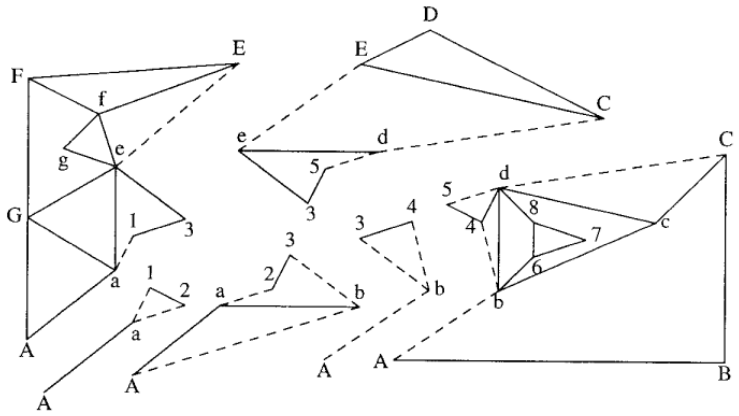


FIG. 12. The slices for the leaves of the tree for the component with nodes 1–5.

way LB and RB were defined in rule $F1$, the slices of the leaves of C include the slices of all the children of u. Thus, in either case, the slice of u includes the slices of its descendants (including the slices of any vertices for enclosed components) and the slices of any vertices corresponding to the component (if any) enclosed by face(u). By induction, the result holds for all vertices at level $j-1$.

PROPOSITION 3. Every edge of the graph is included in the slice of at least one tree vertex.

PROOF. Every edge between two level i nodes, $i > 1$, is the label of some tree vertex and is included in the slice for that vertex by rules $S1$–$S4$.

So consider an edge e between a level i node LL and a level $i-1$ node of the enclosing face. By planarity, e lies between two successive exterior edges e_1 and e_2 incident on u. By rules $F1$, $B2$, and $S4$, e is included in the slice for one of e_1 and e_2.

1.5.3 An overview of the dynamic program

With slices now fully defined, we can start adapting the rest of the algorithm, which mainly breaks down to the dynamic program. Again, we want to compute tables for subgraphs, in this case, smaller slices, and then merge the solutions. The tables will contain the size of the optimal solution for the given assignment of the boundary nodes. As a slice is being constructed according to the definition above, its table will be computed at the same time. This means the order of merges is clearly given by the construction rules of slices.

The dynamic program also uses other subroutines, for example a modified adjust, to deal with single edges being incorporated into the graph or changing the boundaries to the right size so

merges can be performed, but is still based on the merge routine. For that reason, we will now take a deeper look at merge.

As before, merge merges two tables T_1 and T_2 for two subgraphs, now level i slices S_1 and S_2, which share a boundary y. Let the boundary of S_1 be (x, y) and the boundary of S_2 be (y, z). The resulting table corresponds to a slice with boundary (x, z). We can again use the formula $T_1(x, y) + T_2(y, z) - |y|_1$ used for outerplanar graphs.

For every combination of x and z, the maximum over all assignments of y must be computed. Remember that x, y, and z are vectors containing i nodes, so that there are 2^i possible assignments for each of them. Again, the number of nodes that are part of the optimal solution and the merged boundary y must be subtracted to avoid being counted twice.

This operation also dominates the running time of the algorithm. A table for each combination of x and z must be computed, which leads to $2^i \cdot 2^i$ tables. Each computation must consider 2^i possible assignments of y. For a k-outerplanar graph, i is bounded from above by k as i is the level of the considered vertex. A merge is called at most once for every face vertex, as its face or edge is then incorporated into the already solved part of the graph and does not need to be incorporated again. The number of vertices is linear in the number of edges, which is, in planar graphs, linear in the number of nodes. This leads to a total running time of $O(2^k \cdot 2^k \cdot 2^k \cdot n) = O(8^k n)$.

1.5.4 Dynamic Programming Algorithm

We describe procedures adjust, merge, contract, create, and extend informally since the details of the table manipulations are tedious and straightforward.

1. adjust(T): This operation checks the relationship between the two highest level nodes in the boundaries of the slice represented by T and modifies T accordingly. Let the highest level nodes in the left and right boundaries be denoted by x and y, respectively. If x = y, then any entry requiring exactly one of x and y to be in the maximum independent set is set to ”undefined,” and for any entry with both in the independent set, the count of nodes in the independent set is corrected to avoid counting $x = y$ twice. If $x \neq y$ and (x, y) is an edge in the graph, any entry that requires both x and y to be in the set is set to ”undefined.”

2. merge(T1, T2): This operation merges two tables $T1$ and $T2$ for level i slices $S1$ and $S2$, respectively, that share a common boundary, where the right boundary of $S1$ is the same as

Algorithm 2 Procedure table(v)

Let (x, v) be the level of v;
Let v be the level of l;
if t is a face vertex and face(u) encloses no level [+1 component **then**
    T = table(u), where u is the leftmost child of i;
    for each other child c of v' from left to right **do**
        T = merge(T, table(c));
    end for
    return adjust(T);
else if v is a face vertex and face(r) encloses a level $t+1$ component C **then**
    return adjust(contract(table(root(C))));
else if v is a level 1 leaf **then**
    return stable representing the edge (x, v);
else ▷ [is a level i leaf, $z > 1$]
    Let f be the level i face enclosing the component for v;
    Let the labels of the children of vertex(f) be $(z_1, z_2), ..., (z_m, z_{m+1})$;
    if y is adjacent to some z, LB(u) $< r <$ RB(v') **then**
        Let p be the least such r;
    else
        p = RB(v);
    end if
    ▷ Note: p is a point between nodes adjacent to x and nodes adjacent to v
    T = create(c, p);
    ▷ Extend the leftmost p tables to include x and edges from x to r, $r < p$, and merge with T
    $j = p - 1$;
    while $j >$ LB(v) **do**
        T = merge(extend(x, table(u), T)), where u_1 is the jth child of vertex(f);
        $j = j - 1$;
    end while
    ▷ Extend the remaining tables to include v and edges from y to r, $r > p$, and merge with T
    $j = p$;
    while $j <$ RB(v) **do**
        T = merge(T, extend(y, table)), where u_2 is the jth child of vertex(f);
        $j = j + 1$;
    end while
    return T;
end if

the left boundary of $S2$. The resulting table will be for a slice whose first boundary is the left boundary of $S1$ and whose second boundary is the right boundary of $S2$. For each pair of vectors v_1, v_2 representing whether each vertex in the boundaries of the new slice is in the set, the new value will be the maximum over all pairs of the value in $T1$ for v_1, v_2 plus the value in $T2$ for v_2, v_1 minus the number of 1's in v_2 (to avoid counting any vertex twice).

3. contract(T): This operation changes a level $i+1$ table T into a level i table T' (with a shorter boundary). Here, T is the table for root(C), where C is the tree of a level $i+1$ component enclosed by a level-i face F, and T' is the table for vertex(F). Let S = slice(root(C)) and S' = slice(vertex(F)). For some vectors v_1 and v_2, the left and right boundaries, respectively, for slice(vertex(F)) are x, v_1 and y, v_2. By construction, the label of the root of F is (z, z) for some z, and the boundaries of S are z, x, v_1 and z, y, v_2. For each pair of $(0, 1)$-valued vectors representing whether each node of x, v_1, y, and v_2 is in the independent set, T has two values: one for z in the set, and one for z not in the set. "Contract" picks the larger of these two values as the new value for x, v_1 and y, v_2. The resulting table has $2Z'$ entries, reflecting the boundaries of length i of S'.

4. create(v', p): In this case, v is a leaf of a tree for a level $i+1$ component enclosed by a face F, and $p = t+1$, where the children of vertex(F) are $u_1, u_2, ..., u_t$. This operation creates a table for the subgraph including (i) the edge (x, y) represented by v', (ii) the subgraph induced by the left boundary of v', if $p < t$, or the right boundary of v' if $p = t+1$, and (iii) any edges from x or y to the level-i node of this boundary. Since at most $i+2$ nodes and $i+2$ edges can occur in this slice, each entry of the table can be computed in $O(i)$ time.

5. extend(z, T): Given a table T for a level-i slice and a level $i+1$ node z, this operation computes a table for a level $i+1$ slice as follows: The boundaries of the new slice will be the old boundaries plus z. For any vectors v_1, v_2 representing whether each of the boundary points of the level-i slice is in the maximum independent set, the new table has two entries: one for z in the set, and one for z not in the set. For z not in the set, the value in the new table will be the same as the old value for v_1, v_2. For z in the set, the new value is undefined if z and a level-i boundary node are both in the set and are adjacent, and one more than before otherwise.

We claim that the above algorithm produces a correct table for the slice of the root of the level-1 tree, this slice includes the whole graph, and the boundaries of this slice are both a, where (a, a) is the label of the root of the level 1 tree. By definition of tables, the table includes four

values, according to whether each of the boundary nodes is in the independent set. Two of the values are undefined since they represent inconsistency as to whether a is in the set. Taking the best of the remaining two values gives the solution for the maximum independent set.

1.6 Proof of Correctness

LEMMA 1. Calling the main procedure table on the root of the level-1 tree leads to exactly one recursive call on every other vertex of each tree of each level.

PROOF. First, we show by contradiction that table cannot be called more than once on the same tree vertex. For if so, list in order the successive vertices on which table is called, and let u be the first vertex that appears for the second time in the list. If u is the root of a tree, table is called on u only from within a call on vertex(f), where f is the face enclosing the component corresponding to u; since no second call has occurred on vertex(f) by the time the second call occurs for u, u cannot be the root of a tree. Let v be the parent of u. If face(u) does not enclose a higher-level component, then table is called on u only from within a call on u; as before, since v has not had a second call, this case cannot apply. Therefore, face(u) encloses a higher-level component C, and table is called on u only while processing a leaf of C.

Suppose u is the jth child of v. If table is called on u while processing a leaf y of C, then according to the algorithm, $LB(y) \leq j < RB(y)$. But the definitions of LB and RB imply that each is nondecreasing for leaves taken from left to right, and $RB(y) = LB(z)$, where z is the right sibling (if any) of y. Therefore, $j < RB(y) = LB(z)$ and table cannot be called on u while processing any leaf of C to the right of y. We conclude that table is called at most once on u, contradicting the choice of u.

Next, we show that the main procedure is called on every vertex. We do this inductively by showing that for every vertex u, a call on u results in a call on every descendant of u and on every vertex of a tree for a component enclosed by a face corresponding to u or a descendant of u. For a level k vertex v, there are no enclosed components, and the structure of the algorithm causes recursive calls on descendants. Assume that the statement is true for level $j < k$ and for descendants of a level $j-1$ node u. If v is a leaf, the statement is vacuous. So suppose u is not a leaf. If face(u) does not enclose any higher-outerplanar component, we need only apply the induction hypothesis to obtain the desired result. If face(u) encloses a higher-outerplanar component, a call on u results in a call on the root of the tree for this component, and by the induction hypothesis,

this call results in calls on all the leaves of this tree. But calls on the leaves result in calls on the children of v; because of the way LB and RB are defined, a call is made on every child of v. By the induction hypothesis applied to the children of v, the desired result holds.

LEMMA 2. A call of *table* on the root of the level 1 tree results in a correct table for the corresponding slice.

PROOF. We show that a call on a vertex v_j results in a correct table for the slice of v_j. The proof is by induction on the number of recursive calls caused by a call on v. By Lemma 1, the number of recursive calls is always finite. We assume that the procedures *adjust*, *merge*, *extend*, *create*, and *contract* are implemented correctly, and that the algorithm computes a table correctly for a level-1 leaf.

If a call on u does not generate any recursive calls, either v_j is a level 1 leaf, or u is a level-i leaf, $i > 1$, $LB(v') = RB(v')$, and a table is computed by *create*. In either case, the resulting table is correct by assumption.

Assume for j that whenever at most j recursive calls are made a correct table is computed. Suppose the call on a level-i vertex u generates $j + 1$ recursive calls. We show that the table computed for v_j must also be correct.

Let (x, y) be the label of v_j. First, suppose u is a face vertex and face(v_j) encloses no level $i+1$ component. By rule S1, the slice for v_j is the union of the slices of v_j's children plus (x, y) if (x, y) is an edge. Also, the left (right) boundary of the slice of v_j is the same as the left boundary of v_j's leftmost (rightmost) child by Proposition 1 above. After the tables for the children are merged in pairs, the resulting table is for a slice consisting of the union of the children's slices and the left (right) boundary of the slice is derived from the leftmost (rightmost) child. By Proposition 1, the left boundary includes x and the right boundary includes y.

However, at this point, the table assumes that $x \neq y$ and (x, y) is not an edge. It also assumes that all lower-level boundary nodes are distinct, even though they might be duplicated on the left and right boundaries. However, our definition of a level i table requires that any duplicate boundary nodes not at level i be treated as if they are distinct. Therefore, the table is correct except for the treatment of x and y. If $x = y$, *adjust* sets to "undefined" any table entry requiring exactly one of them to be in the independent set and corrects the count in any table entry requiring both to be in the independent set. If $x \neq y$ and (x, y) is an edge, *adjust* sets to "undefined" any entry that requires both x and y to be in the independent set. Thus, the final table computed for

v_j is correct.

Second, suppose u is a face vertex and face(v_j) encloses a level $i+1$ component C. By rule S2, the slice for v' is the subgraph consisting of the slice of $mot(v')$ plus (x, y) if (x, y) is an edge. Also, the left (right) boundary of v_j's slice is the same as the left (right) boundary of v_j's leftmost (rightmost) child by Proposition 1. Now, the left (right) boundary of the slice of root(C) is the same as the left (right) boundary of the slice of the leftmost (rightmost) leaf of C by Proposition 1 applied inductively. Let u' and v' be the leftmost and rightmost leaves of C, respectively. Let (c, d) be the label of the root of C. By the definition of tree labeling, c is the first node in the label of $v_{u'}$ and d is the second node in the label of $v_{v'}$. By rule B1 of the definition of LB and RB, $LB(u') = 1$ and $RB(v') = r + 1$, where r is the number of children of u'. By rule B2 of the definition of boundaries, the left boundary of $v_{u'}$ is c plus the left boundary of the leftmost child of C, and the right boundary of u' is d plus the right boundary of the rightmost child of C. Therefore, the boundaries of v_j's slice are the same as those of the slice of the root of C except for the extra level $i+1$ nodes c and d for the latter slice. Consequently, the table computed by the algorithm for the root of C is correct for u' except for (i) the extra nodes in the boundaries, (ii) the nonincorporation of (x, y) if (x, y) is an edge, and (iii) ignorance of the duplication of x if $x = y$ (since duplication is taken into account only for level-$(i+1)$ nodes). The procedure *contract* corrects for (i). By Proposition 1 and the definition of boundaries, x and y are on the left and right boundaries, respectively of the slices of C and u'. Hence, *adjust* corrects for (ii) and (iii). Therefore, the final table is correct for v_j.

Contents

1 Introduction to Parameterized Algorithms

Contents

1.1 Fixed-Parameter Algorithms

Algorithms with running time $f(k) \cdot n^c$, for a constant c independent of both n and k, are called fixed-parameter algorithms, or FPT algorithms. Typically, the goal in parameterized algorithmics is to design FPT algorithms, aiming to minimize both the $f(k)$ factor and the constant c in the running time bound. FPT algorithms are contrasted with less efficient XP algorithms (for slice-wise polynomial), where the running time is of the form $f(k) \cdot n^{g(k)}$, for some functions f, g. There is a significant disparity in the running times $f(k) \cdot n^{g(k)}$ and $f(k) \cdot n^c$ [3].

In parameterized algorithmics, k serves as a relevant secondary measurement that encapsulates some aspect of the input instance, whether it represents the size of the solution sought after or a parameter describing the ”structured” nature of the input instance.

1.2 Parameterized Problem

A parameterized problem is a language $L \subseteq \Sigma^* \times \mathbb{N}$, where Σ is a fixed, finite alphabet. For an instance $(x, k) \in \Sigma^* \times \mathbb{N}$, k is called the parameter.

For example, an instance of Clique parameterized by the solution size is a pair (G, k), where we expect G to be an undirected graph encoded as a string over Σ, and k is a positive integer. That is, a pair (G, k) belongs to the Clique parameterized language if and only if the string G correctly encodes an undirected graph, denoted by G, and the graph contains a clique on k vertices. Similarly, an instance of the CNF-SAT problem (satisfiability of propositional formulas

in CNF), parameterized by the number of variables, is a pair (ϕ, n), where we expect ϕ to be the input formula encoded as a string over Σ and n to be the number of variables of ϕ. That is, a pair (ϕ, n) belongs to the CNF-SAT parameterized language if and only if the string ϕ correctly encodes a CNF formula with n variables, and the formula is satisfiable.

We define the size of an instance (x, k) of a parameterized problem as $|x|+k$. One interpretation of this convention is that, when given to the algorithm as input, the parameter k is encoded in unary.

1.3 Fixed-Parameter Tractable

A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called fixed-parameter tractable (FPT) if there exists an algorithm A (called a fixed-parameter algorithm), a computable function $f : \mathbb{N} \to \mathbb{N}$, and a constant c such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm A correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^c$. The complexity class containing all fixed-parameter tractable problems is called FPT.

In the context of fixed-parameter tractability and the definition of the complexity class XP, it is important to note that the computability of the function f is a fundamental requirement to avoid complications in the development of complexity theory. Additionally, assuming that f is nondecreasing does not alter the definition of fixed-parameter tractability, as any computable function $f : \mathbb{N} \to \mathbb{N}$ can be replaced by a computable nondecreasing function $\bar{f}$ that is always greater than or equal to f. This assumption is crucial for various scenarios, such as reductions, and aligns with the typical behavior in algorithmic results where the running time bound is a nondecreasing function of the complexity measure. These considerations set the stage for the subsequent definition of the complexity class XP.

1.4 Slice-wise Polynomial

A parameterized problem $L \subseteq \Sigma^* \times \mathbb{N}$ is called slice-wise polynomial (XP) if there exists an algorithm A and two computable functions $f, g : \mathbb{N} \to \mathbb{N}$ such that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, the algorithm A correctly decides whether $(x, k) \in L$ in time bounded by $f(k) \cdot |(x, k)|^{g(k)}$. The complexity class containing all slice-wise polynomial problems is called XP.

2 Kernelization

Contents

2.1 Formal Definitions

In the context of kernelization, a reduction rule for a parameterized problem Q is defined as a function $\phi : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ that transforms an instance (I, k) of Q into an equivalent instance (I', k') of Q, where ϕ is computable in polynomial time with respect to the size of I and k. Two instances of Q are considered equivalent if $(I, k) \in Q$ if and only if $(I', k') \in Q$. This property of the reduction rule ϕ, which ensures the translation to an equivalent instance, is commonly known as the safeness or soundness of the reduction rule.

The objective is to develop a preprocessing algorithm that iteratively applies different reduction rules to minimize the size of the instance. This preprocessing algorithm takes an input instance $(I, k) \in \Sigma^* \times \mathbb{N}$ of Q, operates in polynomial time, and produces an equivalent instance (I', k') of Q. To formalize the condition that the output instance should be small, the main principle of Parameterized Complexity is employed, where the complexity is evaluated based on the parameter. Consequently, the output size of a preprocessing algorithm A is represented by a function $size_A : \mathbb{N} \to \mathbb{N} \cup \{\infty\}$ defined as:

$$size_A(k) = \sup\{|I'| + k' : (I', k') = A(I, k), I \in \Sigma^*\}$$

This function calculates the supremum of the sum of the sizes of I' and k' for all possible instances of Q with a fixed parameter k. It is important to note that this supremum may be infinite if there is no bound on the size of $A(I, k)$ solely in terms of the input parameter k.

2

Kernelization algorithms are precisely those preprocessing algorithms where the output size is finite and constrained by a computable function of the parameter.

Definition 2.1 (Kernelization, kernel). A *kernelization algorithm*, or simply a *kernel*, for a parameterized problem Q is an algorithm A that, given an instance (I, k) of Q, works in polynomial time and returns an equivalent instance (I', k') of Q. Moreover, we require that $size_A(k) \leq g(k)$ for some computable function $g : \mathbb{N} \to \mathbb{N}$.

The size requirement in the kernelization definition can be restated as the existence of a computable function $g(\cdot)$ such that for any output instance (I', k') resulting from an input instance (I, k), it must satisfy $|I'| + k' \leq g(k)$. If the upper bound $g(\cdot)$ is a polynomial or linear function of the parameter, it indicates that problem Q has a polynomial or linear kernel. In practice, the output of a kernelization algorithm is often referred to as the "reduced" equivalent instance, which is also termed a kernel. Additionally, in certain scenarios where problem instances can be decisively resolved as yes or no instances, the kernelization algorithm may return a constant-size trivial yes-instance or no-instance to align with the kernel definition. Notably, the output parameter k' is constrained to be less than or equal to the input parameter k, ensuring the preservation of problem complexity. The efficiency of a kernelization algorithm is primarily assessed by a bound on its output size, with the algorithm being mandated to operate in polynomial time theoretically.

Lemma 2.2. If a parameterized problem Q is FPT then it admits a kernelization algorithm.

Proof. Since Q is FPT, there is an algorithm A deciding if $(I, k) \in Q$ in time $f(k) \cdot |I|^c$ for some computable function f and a constant c. We obtain a kernelization algorithm for Q as follows. Given an input (I, k), the kernelization algorithm runs A on (I, k), for at most $|I|^c + 1$ steps. If it terminates with an answer, use that answer to return either that (I, k) is a yes-instance or that it is a no-instance. If A does not terminate within $|I|^c + 1$ steps, then return (I, k) itself as the output of the kernelization algorithm. Observe that since A did not terminate in $|I|^c + 1$ steps, we have that $f(k) \cdot |I|^c > |I|^c + 1$, and thus $|I| < f(k)$. Consequently, we have $|I| + k \leq f(k) + k$, and we obtain a kernel of size at most $f(k) + k$; note that this upper bound is computable as $f(k)$ is a computable function.

2.2 Some Simple Kernels

2.2.1 Vertex Cover

In the Vertex Cover problem, given a graph G and a positive integer k as input, the goal is to determine if there exists a vertex cover of size at most k. A set S is considered a vertex cover if for every edge of G, at least one of its endpoints is in S, making the graph $G - S$ edgeless and $V(G) \setminus S$ an independent set.

The first reduction rule is based on the observation that if a graph G in the Vertex Cover instance has an isolated vertex, removing this vertex does not affect the solution. This reduction rule is safe.

Reduction VC.1. If G contains an isolated vertex v, delete v from G. The new instance is $(G - v, k)$.

Reduction VC.2. If there is a vertex v of degree at least $k+1$, then delete v (and its incident edges) from G and decrement the parameter k by 1. The new instance is $(G - v, k - 1)$.

Lemma 2.3. If (G, k) is a yes-instance and none of the reduction rules VC.1, VC.2 is applicable to G, then $|V(G)| \leq k^2 + k$ and $|E(G)| \leq k^2$.

Proof. Because we cannot apply Reductions VC.1 anymore on G, G has no isolated vertices. Thus for every vertex cover S of G, every vertex of $G - S$ should be adjacent to some vertex from S. Since we cannot apply Reductions VC.2, every vertex of G has degree at most k. It follows that $|V(G - S)| \leq k|S|$ and hence $|V(G)| \leq (k + 1)|S|$. Since (G, k) is a yes-instance, there is a vertex cover S of size at most k, so $|V(G)| \leq (k + 1)k$. Also, every edge of G is covered by some vertex from a vertex cover and every vertex can cover at most k edges. Hence if G has more than k^2 edges, this is again a no-instance.

Reduction VC.3. Let (G, k) be an input instance such that Reductions VC.1 and VC.2 are not applicable to (G, k). If $k < 0$ and G has more than $k^2 + k$ vertices, or more than k^2 edges, then conclude that we are dealing with a no-instance.

Theorem 2.4. Vertex Cover admits a kernel with $O(k^2)$ vertices and $O(k^2)$ edges.

2.2.2 Feedback Arc Set in Tournaments

Lemma 2.5. A directed graph G is acyclic if and only if it is possible to order its vertices in such a way such that for every directed edge (u, v), we have $u < v$.

Observation 2.6. Let G be a directed graph and let F be a subset of edges of G. If $G \diamond F$ is a directed acyclic graph, then F is a feedback arc set of G.

Lemma 2.7. Let G be a directed graph and F be a subset of $E(G)$. Then F is an inclusion-wise minimal feedback arc set of G if and only if F is an inclusion-wise minimal set of edges such that $G \diamond F$ is an acyclic directed graph.

Proof. We first prove the forward direction of the lemma. Let F be an inclusion-wise minimal feedback arc set of G. Assume to the contrary that $G \diamond F$ has a directed cycle C. Then C cannot contain only edges of $E(G) \setminus F$, as that would contradict the fact that F is a feedback arc set. Let $f_1, f_2, \ldots, f_\ell$ be the edges of $C \cap \mathrm{rev}(F)$ in the order of their appearance on the cycle C, and let $e_i \in F$ be the edge f_i reversed. Since F is inclusion-wise minimal, for every e_i, there exists a directed cycle C_i in G such that $F \cap C_i = \{e_i\}$.

Now consider the following closed walk W in G: we follow the cycle C, but whenever we are to traverse an edge $f_i \in \mathrm{rev}(F)$ (which is not present in G), we instead traverse the path $C_i - e_i$. By definition, W is a closed walk in G and, furthermore, note that W does not contain any edge of F. This contradicts the fact that F is a feedback arc set of G.

The minimality follows from Observation 2.6. That is, every set of edges F such that $G \diamond F$ is acyclic is also a feedback arc set of G, and thus, if F is not a minimal set such that $G \diamond F$ is acyclic, then it will contradict the fact that F is a minimal feedback arc set.

For the other direction, let F be an inclusion-wise minimal set of edges such that $G \diamond F$ is an acyclic directed graph. By Observation 2.6, F is a feedback arc set of G. Moreover, F is an inclusion-wise minimal feedback arc set, because if a proper subset F' of F is an inclusion-wise minimal feedback arc set of G, then by the already proved implication of the lemma, $G \diamond F'$ is an acyclic directed graph, a contradiction with the minimality of F.

Theorem 2.8. Feedback Arc Set in Tournaments admits a kernel with at most $k^2 + 2k$ vertices.

Proof. Lemma 2.7 implies that a tournament T has a feedback arc set of size at most k if and only if it can be turned into an acyclic tournament by reversing directions of at most k edges. We will use this characterization for the kernel.

In what follows by a triangle we mean a directed cycle of length three. We give two simple reduction rules.

Reduction FAST.1. If an edge e is contained in at least $k+1$ triangles, then reverse e and reduce k by 1.

Reduction FAST.2. If a vertex v is not contained in any triangle, then delete v from T.

The rules follow similar guidelines as in the case of Vertex Cover. In Reduction FAST.1, we greedily take into a solution an edge that participates in $k+1$ otherwise disjoint forbidden structures (here, triangles). In Reduction FAST.2, we discard vertices that do not participate in any forbidden structure, and should be irrelevant to the problem.

However, a formal proof of the safeness of Reduction FAST.2 is not immediate: we need to verify that deleting v and its incident edges does not make a yes-instance out of a no-instance.

Note that after applying any of the two rules, the resulting graph is again a tournament. The first rule is safe because if we do not reverse e, we have to reverse at least one edge from each of $k+1$ triangles containing e. Thus e belongs to every feedback arc set of size at most k.

Let us now prove the safeness of the second rule. Let $X = N^+(v)$ be the set of heads of directed edges with tail v and let $Y = N^-(v)$ be the set of tails of directed edges with head v. Because T is a tournament, X and Y is a partition of $V(T) \setminus \{v\}$. Since v is not a part of any triangle in T, we have that there is no edge from X to Y (with head in Y and tail in X). Consequently, for any feedback arc set A_1 of tournament $T[X]$ and any feedback arc set A_2 of tournament $T[Y]$, the set $A_1 \cup A_2$ is a feedback arc set of T. As the reverse implication is trivial (for any feedback arc set A in T, $A \cap E(T[X])$ is a feedback arc set of $T[X]$, and $A \cap E(T[Y])$ is a feedback arc set of $T[Y]$), we have that (T, k) is a yes-instance if and only if $(T - v, k)$ is.

Finally, we show that every reduced yes-instance T, an instance on which none of the presented reduction rules are applicable, has at most $k(k+2)$ vertices. Let A be a feedback arc set of a reduced instance T of size at most k. For every edge $e \in A$, aside from the two endpoints of e, there are at most k vertices that are in triangles containing e — otherwise we would be able to apply Reduction FAST.1. Since every triangle in T contains an edge of A and every vertex of T is in a triangle, we have that T has at most $k(k+2)$ vertices.

Thus, given (T, k) we apply our reduction rules exhaustively and obtain an equivalent instance (T', k'). If T' has more than $k'^2 + k'$ vertices, then the algorithm returns that (T, k) is a no-instance, otherwise we get the desired kernel. This completes the proof of the theorem.

2.2.3 Edge Clique Cover

- **Reduction ECC.1.** Remove isolated vertices.
- **Reduction ECC.2.** If there is an isolated edge uv (a connected component that is just an edge), delete it and decrease k by 1. The new instance is $(G - \{u, v\}, k - 1)$.
- **Reduction ECC.3.** If there is an edge uv whose endpoints have exactly the same closed neighborhood, that is, $N[u] = N[v]$, then delete v. The new instance is $(G - v, k)$.

Theorem 2.9. Edge Clique Cover admits a kernel with at most 2^k vertices.

Proof. We start with the following claim.

Claim: If (G, k) is a reduced yes-instance, on which none of the presented reduction rules can be applied, then $|V(G)| \leq 2^k$.

Proof: Let $C_1, \ldots, C_k$ be an edge clique cover of G. We claim that G has at most 2^k vertices. Targeting a contradiction, let us assume that G has more than 2^k vertices. We assign to each vertex $v \in V(G)$ a binary vector b_v of length k, where bit i, $1 \leq i \leq k$, is set to 1 if and only if v is contained in clique C_i. Since there are only 2^k possible vectors, there must be $u = v \in V(G)$ with $b_u = b_v$. If b_u and b_v are zero vectors, the first rule applies; otherwise, u and v are contained in the same cliques. This means that u and v are adjacent and have the same neighborhood; thus either Reduction ECC.2 or Reduction ECC.3 applies. Hence, if G has more than 2^k vertices, at least one of the reduction rules can be applied to it, which is a contradiction to the initial assumption that G is reduced. This completes the proof of the claim.

The kernelization algorithm works as follows. Given an instance (G, k), it applies Reductions ECC.1, ECC.2, and ECC.3 exhaustively. If the resulting graph has more than 2^k vertices, the kernelization algorithm outputs that the input instance is a no-instance; else it outputs the reduced instance.

2.3 Crown Decomposition

Definition 2.10 (Crown decomposition) A crown decomposition of a graph G is a partitioning of $V(G)$ into three parts C, H, and R, such that:

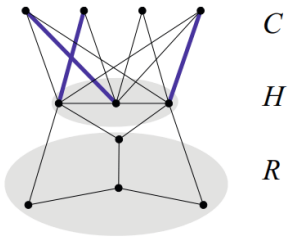


Fig. 2.1: Example of a crown decomposition. Set C is an independent set, H separates C and R, and there is a matching of H into C

1. C is nonempty.

2. C is an independent set.

3. There are no edges between vertices of C and R. That is, H separates C and R.

4. Let E' be the set of edges between vertices of C and H. Then E' contains a matching of size $|H|$. In other words, G contains a matching of H into C.

The set C can be seen as a crown put on head H of the remaining part R, see Fig. 2.1. Note that the fact that E' contains a matching of size $|H|$ implies that there is a matching of H into C. This is a matching in the subgraph G', with the vertex set $C \cup H$ and the edge set E', saturating all the vertices of H.

For finding a crown decomposition in polynomial time, we use the following well-known structural and algorithmic results. The first is a mini-max theorem due to Kőnig.

Theorem 2.11 (Kőnig's theorem) In every undirected bipartite graph, the size of a maximum matching is equal to the size of a minimum vertex cover.

Theorem 2.12 (Hall's theorem) Let G be an undirected bipartite graph with bipartition (V_1, V_2). The graph G has a matching saturating V_1 if and only if for all $X \subseteq V_1$, we have $|N(X)| \geq |X|$.

Theorem 2.13 (Hopcroft-Karp algorithm) Let G be an undirected bipartite graph with bipartition V_1 and V_2, on n vertices and m edges. Then we can find a maximum matching as well as a minimum vertex cover of G in time $O(m\sqrt{n})$. Furthermore, in time $O(m\sqrt{n})$ either we can find a matching saturating V_1 or an inclusion-wise minimal set $X \subseteq V_1$ such that $|N(X)| < |X|$.

Lemma 2.14 (Crown lemma) Let G be a graph without isolated vertices and with at least $3k + 1$ vertices. There is a polynomial-time algorithm that either

- finds a matching of size $k+1$ in G; or
- finds a crown decomposition of G.

Proof. We first find an inclusion-maximal matching M in G. This can be done by a greedy algorithm. If the size of M is $k+1$, then we are done. Hence, we assume that $|M| \leq k$, and let V_M be the endpoints of M. We have $|V_M| \leq 2k$. Because M is a maximal matching, the remaining set of vertices $I = V(G) \setminus V_M$ is an independent set.

Consider the bipartite graph G_{I,V_M} formed by edges of G between V_M and I. We compute a minimum-sized vertex cover X and a maximum sized matching M' of the bipartite graph G_{I,V_M} in polynomial time using Theorem 2.13. We can assume that $|M'| \leq k$, for otherwise we are done. Since $|X| = |M'|$ by Kőnig's theorem (Theorem 2.11), we infer that $|X| \leq k$.

If no vertex of X is in V_M, then $X \subseteq I$. We claim that $X = I$. For a contradiction assume that there is a vertex $w \in I \setminus X$. Because G has no isolated vertices, there is an edge, say wz, incident to w in G_{I,V_M}. Since G_{I,V_M} is bipartite, we have that $z \in V_M$. However, X is a vertex cover of G_{I,V_M} such that $X \cap V_M = \emptyset$, which implies that $w \in X$. This is contrary to our assumption that $w \notin X$, thus proving that $X = I$. But then $|I| \leq |X| \leq k$, and G has at most $|I| + |V_M| \leq k + 2k = 3k$ vertices, which is a contradiction.

Hence, $X \cap V_M = \emptyset$. We obtain a crown decomposition (C, H, R) as follows. Since $|X| = |M'|$, every edge of the matching M' has exactly one endpoint in X. Let M^* denote the subset of M' such that every edge from M^* has exactly one endpoint in $X \cap V_M$ and let V_M^* denote the set of endpoints of edges in M^*. We define head $H = X \cap V_M = X \cap V_M^*$, crown $C = V_M^* \cap I$, and the remaining part $R = V(G) \setminus (C \cup H) = V(G) \setminus V_M^*$. In other words, H is the set of endpoints of edges of M^* that are present in V_M and C is the set of endpoints of edges of M^* that are present in I. Obviously, C is an independent set and by construction, M^* is a matching of H into C. Furthermore, since X is a vertex cover of G_{I,V_M}, every vertex of C can be adjacent only to vertices of H and thus H separates C and R. This completes the proof.

2.3.1 Vertex Cover

In a Vertex Cover instance (G, k), after applying Reduction VC.1 to ensure that G has no isolated vertices, if $|V(G)| > 3k$, we can utilize the crown lemma on the graph G and integer k. This results in either finding a matching of size $k+1$, or a crown decomposition $V(G) = C \cup H \cup R$.

If a matching of size $k+1$ is found, the algorithm determines that (G, k) is a no-instance. In the case of a crown decomposition, let M be a matching of H into C. The matching M implies that for any vertex cover X of G, X must contain at least $|M| = |H|$ vertices from $H \cup C$ to cover the edges of M. Additionally, the set H covers all edges of G incident to $H \cup C$. Therefore, there exists a minimum vertex cover of G that includes H, allowing us to reduce (G, k) to $(G - H, k - |H|)$. It is important to note that in the instance $(G - H, k - |H|)$, the vertices of C become isolated and will be reduced by Reduction VC.1.

Since the crown lemma guarantees that $H = \emptyset$, we can iteratively reduce the graph as long as $|V(G)| > 3k$, leading to the conclusion that Vertex Cover admits a kernel with at most $3k$ vertices.

Theorem 2.15. Vertex Cover admits a kernel with at most $3k$ vertices.

2.3.2 Maximum Satisfiability

Theorem 2.16. Maximum Satisfiability admits a kernel with at most k variables and $2k$ clauses.

Proof: Let ϕ be a CNF formula with n variables and m clauses. Let ψ be an arbitrary assignment to the variables and let $\neg\psi$ be the assignment obtained by complementing the assignment of ψ. That is, if ψ assigns $\delta \in \{T, F\}$ to some variable x, then $\neg\psi$ assigns $\neg\delta$ to x. Observe that either ψ or $\neg\psi$ satisfies at least $m/2$ clauses, since every clause is satisfied by ψ or $\neg\psi$ (or by both). This means that, if $m \geq 2k$, then (ϕ, k) is a yes-instance. In what follows we give a kernel with $n < k$ variables.

Let G_ϕ be the variable-clause incidence graph of ϕ. That is, G_ϕ is a bipartite graph with bipartition (X, Y), where X is the set of the variables of ϕ and Y is the set of clauses of ϕ. In G_ϕ there is an edge between a variable $x \in X$ and a clause $c \in Y$ if and only if either x, or its negation, is in c. If there is a matching of X into Y in G_ϕ, then there is a truth assignment satisfying at least $|X|$ clauses: we can set each variable in X in such a way that the clause matched to it becomes satisfied. Thus at least $|X|$ clauses are satisfied. Hence, in this case, if $k \leq |X|$, then (ϕ, k) is a yes-instance. Otherwise, $k > |X| = n$, and we get the desired kernel.

We now show that, if ϕ has at least $n \geq k$ variables, then we can, in polynomial time, either reduce ϕ to an equivalent smaller instance, or find an assignment to the variables satisfying at least k clauses (and conclude that we are dealing with a yes-instance). Suppose ϕ has at least k variables. Using Hall's theorem and a polynomial-time algorithm computing a maximum-size matching, we can in polynomial time find either a matching of X into Y or an inclusion-wise

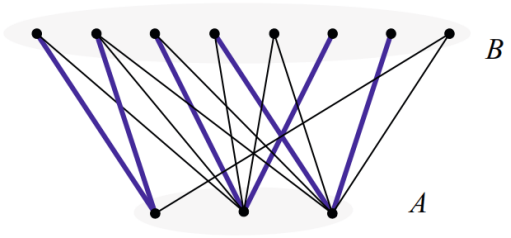


Fig. 2.2: Set A has a 2-expansion into B

minimal set $C \subseteq X$ such that $|N(C)| < |C|$. As discussed in the previous paragraph, if we found a matching, then the instance is a yes-instance and we are done. So suppose we found a set C as described. Let H be $N(C)$ and $R = V(G_\phi) \setminus (C \cup H)$. Clearly, $N(C) \subseteq H$, there are no edges between vertices of C and R, and $G[C]$ is an independent set. Select an arbitrary $x \in C$. We have that there is a matching of $C \setminus \{x\}$ into H since $|N(C')| \geq |C'|$ for every $C' \subseteq C \setminus \{x\}$. Since $|C| > |H|$, we have that the matching from $C \setminus \{x\}$ to H is in fact a matching of H into C. Hence (C, H, R) is a crown decomposition of G_ϕ.

We prove that all clauses in H are satisfied in every assignment satisfying the maximum number of clauses. Indeed, consider any assignment ψ that does not satisfy all clauses in H. Fix any variable $x \in C$. For every variable y in $C \setminus \{x\}$ set the value of y so that the clause in H matched to y is satisfied. Let ψ' be the new assignment obtained from ψ in this manner. Since $N(C) \subseteq H$ and ψ' satisfies all clauses in H, more clauses are satisfied by ψ' than by ψ. Hence ψ cannot be an assignment satisfying the maximum number of clauses. The argument above shows that (ϕ, k) is a yes-instance to Maximum Satisfiability if and only if $(\phi \setminus H, k - |H|)$ is. This gives rise to the following simple reduction.

Reduction MSat.1. Let (ϕ, k) and H be as above. Then remove H from ϕ and decrease k by $|H|$. That is, $(\phi \setminus H, k - |H|)$ is the new instance. Repeated applications of Reduction MSat.1 and the arguments described above give the desired kernel. This completes the proof of the theorem.

2.4 Expansion lemma

Definition: A q-star, $q \geq 1$, is a graph with $q+1$ vertices, one vertex of degree q called the center, and all other vertices of degree 1 adjacent to the center.

Let G be a bipartite graph with vertex bipartition (A, B). For a positive integer q, a set of edges $M \subseteq E(G)$ is called a q-expansion of A into B if:

- Every vertex of A is incident to exactly q edges of M.
- M saturates exactly $q|A|$ vertices in B.

Lemma 2.17. Let G be a bipartite graph with bipartition (A, B). Then there is a q-expansion from A into B if and only if $|N(X)| \geq q|X|$ for every $X \subseteq A$. Furthermore, if there is no q-expansion from A into B, then a set $X \subseteq A$ with $|N(X)| < q|X|$ can be found in polynomial time.

Proof. If A has a q-expansion into B, then trivially $|N(X)| \geq q|X|$ for every $X \subseteq A$.

For the opposite direction, we construct a new bipartite graph G' with bipartition (A', B) from G by adding $(q-1)$ copies of all the vertices in A. For every vertex $v \in A$, all copies of v have the same neighborhood in B as v. We aim to prove that there is a matching M from A' into B in G'. If we prove this, then by identifying the endpoints of M corresponding to the copies of vertices from A, we obtain a q-expansion in G. It suffices to check that the assumptions of Hall's theorem are satisfied in G'. Assume otherwise, that there is a set $X \subseteq A'$ such that $|N_{G'}(X)| < |X|$. Without loss of generality, we can assume that if X contains some copy of a vertex v, then it contains all the copies of v, since including all the remaining copies increases $|X|$ but does not change $|N_{G'}(X)|$. Hence, the set X in A' naturally corresponds to the set X_A of size $|X|/q$ in A, the set of vertices whose copies are in X. But then $|N_G(X_A)| = |N_{G'}(X)| < |X| = q|X_A|$, which is a contradiction. Hence A' has a matching into B and thus A has a q-expansion into B.

For the algorithmic claim, note that if there is no q-expansion from A into B, then we can use Theorem 2.13 to find a set $X \subseteq A'$ such that $|N_{G'}(X)| < |X|$, and the corresponding set X_A satisfies $|N_G(X_A)| < q|X_A|$.

Lemma 2.18. (Expansion lemma) Let $q \geq 1$ be a positive integer and G be a bipartite graph with vertex bipartition (A, B) such that

1. $|B| \geq q|A|$, and
2. there are no isolated vertices in B.

Then there exist nonempty vertex sets $X \subseteq A$ and $Y \subseteq B$ such that

- there is a q-expansion of X into Y, and
- no vertex in Y has a neighbor outside X, that is, $N(Y) \subseteq X$.

Furthermore, the sets X and Y can be found in time polynomial in the size of G. Note that the sets X, Y, and $V(G) \setminus (X \cup Y)$ form a crown decomposition of G with a stronger property — every vertex of X is not only matched into Y, but there is a q-expansion of X into Y. We proceed with the proof of the expansion lemma.

Proof. We proceed recursively, at every step decreasing the cardinality of A. When $|A| = 1$, the claim holds trivially by taking $X = A$ and $Y = B$. We apply Lemma 2.17 to G. If A has a q-expansion into B, then we are done as we may again take $X = A$ and $Y = B$. Otherwise, we can in polynomial time find a (nonempty) set $Z \subseteq A$ such that $|N(Z)| < q|Z|$. We construct the graph G' by removing Z and $N(Z)$ from G. We claim that G' satisfies the assumptions of the lemma. Indeed, because we removed less than q times more vertices from B than from A, we have that (i) holds for G'. Moreover, every vertex from $B \setminus N(Z)$ has no neighbor in Z, and thus (ii) also holds for G'. Note that $Z = A$, because otherwise $N(A) = B$ (there are no isolated vertices in B) and $|B| \geq q|A|$. Hence, we recurse on the graph G' with bipartition $(A \setminus Z, B \setminus N(Z))$, obtaining nonempty sets $X \subseteq A \setminus Z$ and $Y \subseteq B \setminus N(Z)$ such that there is a q-expansion of X into Y and such that $N_{G'}(Y) \subseteq X$. Because $Y \subseteq B \setminus N(Z)$, we have that no vertex in Y has a neighbor in Z. Hence, $N_{G'}(Y) = N_G(Y) \subseteq X$ and the pair (X, Y) satisfies all the required properties.

2.5 Kernels based on linear programming

Theorem 2.19 (Nemhauser-Trotter theorem). There is a minimum vertex cover S of G such that $V_1 \subseteq S \subseteq V_1 \cup V_{1/2}$.

Proof. Let $S^* \subseteq V(G)$ be a minimum vertex cover of G. Define $S = (S^* \setminus V_0) \cup V_1$.

By the constraints of (2.2), every vertex of V_0 can have a neighbor only in V_1 and thus S is also a vertex cover of G. Moreover, $V_1 \subseteq S \subseteq V_1 \cup V_{1/2}$. It suffices to show that S is a minimum vertex cover. Assume the contrary, i.e., $|S| > |S^*|$. Since $|S| = |S^*| - |V_0 \cap S^*| + |V_1 \setminus S^*|$ we infer that $|V_0 \cap S^*| < |V_1 \setminus S^*|$.

Let us define $\varepsilon = \min\{|x_v - \frac{1}{2}| : v \in V_0 \cup V_1\}$.

We decrease the fractional values of vertices from $V_1 \setminus S^*$ by ε and increase the values of vertices from $V_0 \cap S^*$ by ε. In other words, we define a vector $(y_v)_{v \in V(G)}$ as $y_v = \begin{cases} x_v - \varepsilon & \text{if } v \in V_1 \setminus S^*, \\ x_v + \varepsilon & \text{if } v \in V_0 \cap S^*, \\ x_v & \text{otherwise.} \end{cases}$

Note that $\varepsilon > 0$, because otherwise $V_0 = V_1 = \emptyset$, a contradiction with (2.3). This, together with (2.3), implies that $\sum_{v \in V(G)} y_v < \sum_{v \in V(G)} x_v$.

Now we show that $(y_v)_{v \in V(G)}$ is a feasible solution, i.e., it satisfies the constraints of LPVC(G). Since $(x_v)_{v \in V(G)}$ is a feasible solution, by the definition of ε we get $0 \leq y_v \leq 1$ for every $v \in V(G)$. Consider an arbitrary edge $uv \in E(G)$. If none of the endpoints of uv belong to $V_1 \setminus S^*$, then both $y_u \geq x_u$ and $y_v \geq x_v$, so $y_u + y_v \geq x_u + x_v \geq 1$. Otherwise, by symmetry we can assume that $u \in V_1 \setminus S^*$, and hence $y_u = x_u - \varepsilon$. Because S^* is a vertex cover, we have that $v \in S^*$. If $v \in V_0 \cap S^*$, then $y_u + y_v = x_u - \varepsilon + x_v + \varepsilon = x_u + x_v \geq 1$. Otherwise, $v \in (V_{1/2} \cup V_1) \cap S^*$. Then $y_v \geq x_v \geq \frac{1}{2}$. Note also that $x_u - \varepsilon \geq \frac{1}{2}$ by the definition of ε. It follows that $y_u + y_v = x_u - \varepsilon + y_v \geq \frac{1}{2} + \frac{1}{2} = 1$.

Thus $(y_v)_{v \in V(G)}$ is a feasible solution of LPVC(G) and hence (2.4) contradicts the optimality of $(x_v)_{v \in V(G)}$.

Reduction VC.4. Let $(x_v)_{v \in V(G)}$ be an optimum solution to LPVC(G) in a Vertex Cover instance (G, k) and let V_0, V_1, and $V_{1/2}$ be defined as above. If $\sum_{v \in V(G)} x_v > k$, then conclude that we are dealing with a no-instance. Otherwise, greedily take into the vertex cover the vertices of V_1. That is, delete all vertices of $V_0 \cup V_1$, and decrease k by $|V_1|$.

Lemma 2.20. Reduction VC.4 is safe.

Proof. Clearly, if (G, k) is a yes-instance, then an optimum solution to LPVC(G) is of cost at most k. This proves the correctness of the step if we conclude that (G, k) is a no-instance.

Let $G' = G - (V_0 \cup V_1) = G[V_{1/2}]$ and $k' = k - |V_1|$. We claim that (G, k) is a yes-instance of Vertex Cover if and only if (G', k') is. By Theorem 2.19, we know that G has a vertex cover S of size at most k such that $V_1 \subseteq S \subseteq V_1 \cup V_{1/2}$. Then $S' = S \cap V_{1/2}$ is a vertex cover in G' and the size of S' is at most $k - |V_1| = k'$.

For the opposite direction, let S' be a vertex cover in G'. For every solution of LPVC(G), every edge with an endpoint from V_0 should have an endpoint in V_1. Hence, $S = S' \cup V_1$ is a vertex cover in G and the size of this vertex cover is at most $k' + |V_1| = k$.

Theorem 2.21. Vertex Cover admits a kernel with at most $2k$ vertices.

Proof. Let (G, k) be an instance of Vertex Cover. We solve LPVC(G) in polynomial time, and apply Reduction VC.4 to the obtained solution $(x_v)_{v \in V(G)}$, either concluding that we are dealing with a no-instance or obtaining an instance (G', k'). Lemma 2.20 guarantees the safeness of the reduction. For the size bound, observe that $|V(G')| = |V_{1/2}| = \sum_{v \in V_{1/2}} 2x_v \leq 2 \sum_{v \in V(G)} x_v \leq 2k$.

Lemma 2.22. For a graph G with n vertices and m edges, the optimal (fractional) solution to the linear program LPVC(G) can be found in time $O(m\sqrt{n})$.

Proof. We reduce the problem of solving LPVC(G) to a problem of finding a minimum-size vertex cover in the following bipartite graph H. Its vertex set consists of two copies V_1 and V_2 of the vertex set of G. Thus, every vertex $v \in V(G)$ has two copies $v_1 \in V_1$ and $v_2 \in V_2$ in H. For every edge $uv \in E(H)$, we have edges u_1v_2 and v_1u_2 in H.

Using the Hopcroft-Karp algorithm (Theorem 2.13), we can find a minimum vertex cover S of H in time $O(m\sqrt{n})$. We define a vector $(x_v)_{v\in V(G)}$ as follows: if both vertices v_1 and v_2 are in S, then $x_v = 1$. If exactly one of the vertices v_1 and v_2 is in S, we put $x_v = \frac{1}{2}$. We put $x_v = 0$ if none of the vertices v_1 and v_2 are in S. Thus $\sum_{v\in V(G)} x_v = \frac{|S|}{2}$.

Since S is a vertex cover in H, we have that for every edge $uv \in E(G)$ at least two vertices from $\{u_1, u_2, v_1, v_2\}$ should be in S. Thus $x_u + x_v \geq 1$ and vector $(x_v)_{v\in V(G)}$ satisfies the constraints of LPVC(G).

To show that $(x_v)_{v\in V(G)}$ is an optimal solution of LPVC(G), we argue as follows. Let $(y_v)_{v\in V(G)}$ be an optimal solution of LPVC(G). For every vertex v_i, $i \in \{1, 2\}$, of H, we assign the weight $w(v_i) = y_v$. This weight assignment is a fractional vertex cover of H, i.e., for every edge $v_1u_2 \in E(H)$, $w(v_1) + w(u_2) \geq 1$. We have that $\sum_{v\in V(G)} y_v = \frac{1}{2}\sum_{v\in V(G)}(w(v_1) + w(v_2))$.

On the other hand, the value $\sum_{v\in V(H)} w(v)$ of any fractional solution of LPVC(H) is at least the size of a maximum matching M in H. By Kőnig's theorem (Theorem 2.11), $|M| = |S|$. Hence $\sum_{v\in V(G)} y_v = \frac{1}{2}\sum_{v\in V(G)}(w(v_1) + w(v_2)) = \frac{1}{2}\sum_{v\in V(H)} w(v) \geq \frac{|S|}{2} = \sum_{v\in V(G)} x_v$. Thus $(x_v)_{v\in V(G)}$ is an optimal solution of LPVC(G).

Corollary 2.23. For a graph G with n vertices and m edges, the kernel of Theorem 2.21 can be found in time $O(m\sqrt{n})$.

The following proposition is another interesting consequence of the proof of Lemma 2.22.

Proposition 2.24. Let G be a graph on n vertices and m edges. Then LPVC(G) has a half-integral optimal solution, i.e., all variables have values in the set $\{0, \frac{1}{2}, 1\}$. Furthermore, we can find a half-integral optimal solution in time $O(m\sqrt{n})$.

3 Bounded Search Trees

Contents

3.1 Introduction

Let I be an instance of a minimization problem (such as Vertex Cover). We associate a measure $\mu(I)$ with the instance I, which, in the case of FPT algorithms, is usually a function of k alone. In a branch step we generate from I simpler instances $I_1, \ldots, I_{\mathcal{C}} (\mathcal{C} \geq 2)$ of the same problem such that the following hold:

- Every feasible solution S of I_i, $i \in \{1, \ldots, \mathcal{C}\}$, corresponds to a feasible solution $h_i(S)$ of I. Moreover, the set $\{h_i(S) : 1 \leq i \leq \mathcal{C}$ and S is a feasible solution of $I_i\}$ contains at least one optimum solution for I. Informally speaking, a branch step splits problem I into subproblems $I_1, \ldots, I_{\mathcal{C}}$, possibly taking some (formally justified) greedy decisions.
- The number $\mathcal{C}$ is small, e.g., it is bounded by a function of $\mu(I)$ alone.
- Furthermore, for every I_i, $i \in \{1, \ldots, \mathcal{C}\}$, we have that $\mu(I_i) \leq \mu(I) - c$ for some constant $c > 0$. In other words, in every branch we substantially simplify the instance at hand.

In a branching algorithm, we recursively apply branching steps to instances $I_1, I_2, \ldots, I_{\mathcal{C}}$, until they become simple or even trivial. Thus, we may see an execution of the algorithm as a search tree, where each recursive call corresponds to a node: the calls on instances $I_1, I_2, \ldots, I_{\mathcal{C}}$ are children of the call on instance I. The second and third conditions allow us to bound the number of nodes in this search tree, assuming that the instances with non-positive measure are simple. Indeed, the third condition allows us to bound the depth of the search tree in terms of the measure of the original instance, while the second condition controls the number of branches below every node.

Because of these properties, search trees of this kind are often called bounded search trees. A branching algorithm with a cleverly chosen branching step often offers a drastic improvement over a straightforward exhaustive search.

3.2 Vertex Cover

3

Proposition 3.1. Vertex Cover can be solved optimally in polynomial time when the maximum degree of a graph is at most 2. Thus, we branch only on the vertices of degree at least 3, which immediately brings us to the following upper bound on the number of leaves in a search tree:

$$T(k) = \begin{cases} T(k-1) + T(k-3) & \text{if } k \geq 3, \\ 1 & \text{otherwise.} \end{cases}$$

Again, an upper bound of the form $c \cdot \lambda^k$ for the above recursive function can be obtained by finding the largest root of the polynomial equation $\lambda^3 = \lambda^2 + 1$. Using standard mathematical techniques (and/or symbolic algebra packages), the root is estimated to be at most 1.4656. Combined with kernelization, this gives us the following theorem.

Theorem 3.2. Vertex Cover can be solved in time $O(n\sqrt{m} + 1.4656^k k^{O(1)})$.

3.3 Feedback Vertex Set

Reduction FVS.1. If there is a loop at a vertex v, delete v from the graph and decrease k by 1. Moreover, notice that the multiplicity of a multiple edge does not influence the set of feasible solutions to the instance (G, k).

Reduction FVS.2. If there is an edge of multiplicity larger than 2, reduce its multiplicity to 2.

We now reduce vertices of low degree. Any vertex of degree at most 1 does not participate in any cycle in G, so it can be deleted.

Reduction FVS.3. If there is a vertex v of degree at most 1, delete v.

Concerning vertices of degree 2, observe that, instead of including into the solution any such vertex, we may as well include one of its neighbors. This leads us to the following reduction.

Reduction FVS.4. If there is a vertex v of degree 2, delete v and connect its two neighbors by a new edge.

We remark that after exhaustively applying these four reduction rules, the resulting graph G:

(P1) contains no loops,

(P2) has only single and double edges, and

(P3) has minimum vertex degree at least 3.

Moreover, all rules are trivially applicable in polynomial time. From now on, we assume that in the input instance (G, k), graph G satisfies properties (P1)–(P3).

We remark that for the algorithm in this section, we do not need properties (P1) and (P2). However, we will need these properties later for the kernelization algorithm in Section 9.1.

Finally, we need to add a rule that stops the algorithm if we already exceeded our budget.

Reduction FVS.5. If $k < 0$, terminate the algorithm and conclude that (G, k) is a no-instance.

Lemma 3.3. Every feedback vertex set in G of size at most k contains at least one vertex of V_{3k}.

Proof. To prove this lemma we need the following simple claim.

Claim 3.4. For every feedback vertex set X of G, $\sum_{v\in X}(d(v)-1) \geq |E(G)| - |V(G)| + 1$.

Proof. Graph $F = G - X$ is a forest and thus the number of edges in F is at most $|V(G)|-|X|-1$. Every edge of $E(G) \setminus E(F)$ is incident to a vertex of X. Hence $\sum_{v\in X} d(v) + |V(G)| - |X| - 1 \geq |E(G)|$.

Targeting a contradiction, let us assume that there is a feedback vertex set X of size at most k such that $X \cap V_{3k} = \emptyset$. By the choice of V_{3k}, for every $v \in X$, $d(v)$ is at most the minimum of vertex degrees from V_{3k}. Because $|X| \leq k$, by Claim 3.4 we have that $\sum_{i=1}^{3k}(d(v_i) - 1) \geq 3 \cdot \left(\sum_{v\in X}(d(v) - 1)\right) \geq 3 \cdot (|E(G)| - |V(G)| + 1)$.

In addition, we have that $X \subseteq V(G) \setminus V_{3k}$, and hence $\sum_{i>3k}(d(v_i) - 1) \geq \sum_{v\in X}(d(v) - 1) \geq (|E(G)| - |V(G)| + 1)$.

Therefore, $\sum_{i=1}^{n}(d(v_i) - 1) \geq 4 \cdot (|E(G)| - |V(G)| + 1)$.

However, observe that $\sum_{i=1}^{n} d(v_i) = 2|E(G)|$: every edge is counted twice, once for each of its endpoints. Thus we obtain $4 \cdot (|E(G)| - |V(G)| + 1) \leq \sum_{i=1}^{n}(d(v_i) - 1) = 2|E(G)| - |V(G)|$, which implies that $2|E(G)| < 3|V(G)|$. However, this contradicts the fact that every vertex of G is of degree at least 3.

Theorem 3.5. There exists an algorithm for Feedback Vertex Set running in time $(3k)k \cdot n^{O(1)}$.

Proof. Given an undirected graph G and an integer $k \geq 0$, the algorithm works as follows. It first applies Reductions FVS.1, FVS.2, FVS.3, FVS.4, and FVS.5 exhaustively. As a result, we either already conclude that we are dealing with a no-instance, or obtain an equivalent instance (G', k') such that G' has minimum degree at least 3 and $k' \leq k$. If G' is empty, then we conclude that we are dealing with a yes-instance, as $k' \geq 0$ and an empty set is a feasible solution. Otherwise, let $V_{3k'}$ be the set of $3k'$ vertices of G' with largest degrees. By Lemma 3.3, every solution X to the Feedback Vertex Set instance (G', k') contains at least one vertex from $V_{3k'}$. Therefore, we branch on the choice of one of these vertices, and for every vertex $v \in V_{3k'}$, we recursively apply the algorithm to solve the Feedback Vertex Set instance $(G' - v, k' - 1)$. If one of these branches returns a solution X', then clearly $X' \cup \{v\}$ is a feedback vertex set of size at most k' for G'. Else, we return that the given instance is a no-instance.

At every recursive call we decrease the parameter by 1, and thus the height of the search tree does not exceed k'. At every step we branch in at most $3k'$ subproblems. Hence the number of nodes in the search tree does not exceed $(3k')^{k'} \leq (3k)^k$. This concludes the proof.

Conclusion

In conclusion, the Summer Research Fellowship 2024 at Indian Institute of Technology Madras has been a remarkable journey of exploration and learning in the realm of Parameterized Algorithms and Kernelization. Through the dedicated supervision of Dr. Akanksha Agrawal and the conducive research environment at the RAnG Lab, I have delved deep into the intricacies of computational problem-solving, particularly in the domains of Algorithms & Graphs.

This research experience has not only expanded my knowledge but has also honed my analytical skills and research acumen. The exposure to advanced concepts and methodologies in computer science has been both challenging and rewarding, pushing the boundaries of my understanding and igniting a passion for innovative problem-solving approaches.

As I reflect on the insights gained during this fellowship, I am filled with gratitude towards Dr. Akanksha, and the entire research community at the institute for their unwavering support and encouragement. The lessons learned and the experiences garnered will undoubtedly shape my future academic pursuits and professional endeavors in the dynamic field of Computer Science and Engineering.

Moving forward, I am excited to apply the knowledge and skills acquired during this fellowship to contribute meaningfully to the ongoing advancements in Algorithms & Graphs. This journey has been transformative, and I am eager to continue exploring new frontiers in research and innovation, inspired by the rich tapestry of knowledge woven during this enriching summer research experience.

Bibliography

[1] B. S. Baker, "Approximation algorithms for NP-complete problems on planar graphs," *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pp. 265–273, 1983.

[2] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms, fourth edition*. MIT Press, 2022, ISBN: 9780262046305.

[3] M. Cygan, F. V. Fomin, Ł. Kowalik, *et al.*, *Parameterized algorithms*. Springer, 2015, vol. 5.

[4] J. Kleinberg and É. Tardos, *Algorithm Design*. Pearson/Addison-Wesley, 2006, ISBN: 9780321295354.