FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Advanced Algorithms & Data Structures Course Exam Notes

Emil Petersen, pgx817

2019/05/16

Contents

1 Max Flow **3**
- 1.1 Disposition . . . 3
- 1.2 Presentation . . . 4

2 Linear Programming and Optimization **7**
- 2.1 Disposition . . . 7
- 2.2 Presentation . . . 8

3 Randomized Algorithms **10**
- 3.1 Disposition . . . 10
- 3.2 Comprehensive Notes . . . 11
  - 3.2.1 RandQS Algorithm . . . 11
  - 3.2.2 Min-Cut Algorithm . . . 13
  - 3.2.3 Las Vegas and Monte Carlo . . . 14
- 3.3 Presentation . . . 15

4 Hashing **17**
- 4.1 Disposition . . . 17
- 4.2 Comprehensive Notes (Mostly HSHIS Paper) . . . 18
  - 4.2.1 Hashing Schemes . . . 19
- 4.3 Presentation . . . 21

5 van Emde Boas Trees **22**
- 5.1 Disposition . . . 22
- 5.2 Comprehensive Notes . . . 23
- 5.3 Presentation . . . 25

6 NP Completeness **26**
- 6.1 Disposition . . . 26
- 6.2 Comprehensive Notes (Mostly CW lecture slides) . . . 27
  - 6.2.1 Showing that `VERTEX-COVER` is NP-complete . . . 30
  - 6.2.2 Showing that `TSP` is NP-complete . . . 31
- 6.3 Short Notes . . . 32

7 Exact Exponential Algorithms and Parameterized Complexity **33**
- 7.1 Disposition . . . 33
- 7.2 Comprehensive Notes . . . 34
  - 7.2.1 Exact Exponential Algorithms (EEA) . . . 34
  - 7.2.2 Parameterized Complexity (PC) . . . 34
  - 7.2.3 Dynamic Programming for TSP . . . 35
  - 7.2.4 Bar-Fight Prevention a.k.a Vertex Cover . . . 36
- 7.3 Presentation . . . 36

8 Approximation Algorithms **37**
- 8.1 Disposition . . . 37
- 8.2 Comprehensive Notes . . . 38
  - 8.2.1 The Vertex-Cover Problem . . . 39
  - 8.2.2 Randomized Approximation Algorithm for MAX-3-CNF Satisfiability . . . 40

8.3 Presentation 41

References **42**

Algorithms is about finding scalable solutions to computational problems, and the reliance is only increasing as we enter the world of Big Data. We want algorithms that solve problems efficiently relative to the input size. Exponential time is hopeless. We generally want polynomial time, and for large problems we need linear time. Sometimes we employ data structures that represent the input so that queries about it can be answered very efficiently. In this mandatory course, we will study the list of algorithmic topics below. Some of these topics are covered in more depth in more specialized elective courses.

The book is CLRS [1]. The notes are incomplete.

The (2018) curriculum

Max Flow: CLRS: 26.1, 26.2, 26.3, Theorem 26.6, Edmonds-Karp algorithm and proof of its running time.

Linear Programming: CLRS 29.1, 29.2, 29.3 (until, and excluding, the paragraph "Pivoting"), 29.4 (until Theorem 29.10, but excluding its proof).

Randomized Algorithms: Motwani & Raghavan Chapter 1, beginning, section 1.1 and 1.2.

Hashing: "High Speed Hashing for Integers and Strings" by Thorup, sections 1, 2 and part of 3, up to and including Section 3.3. Proofs that the concrete schemes (multiply mod prime or shift) are (strongly) universal are excluded. The proof of (11) is also excluded although you should understand its relevance to coordinated sampling.

van Emde Boas Trees: CLRS Chapter 20.

NP-completeness: CLRS, Chapter 34, sections 34.1, 34.2, 34.3, 34.4 and 34.5, except the proof of Theorem 34.13).

Exact exponential algorithms and parameterized complexity: For exact exponential algorithms we use Fomin & Kratsch, chapter 1 (travelling salesman and maximum independent set). For parameterized complexity we use chapter 1 from "Parameterized Algorithms".

Approximation algorithms: CLRS Chapter 35, 35.1, 35.2, 35.3, except section 35.2.2, 35.4 and 35.5.

Delaunay triangulations: *Cursorily.* Chapter 9 from Marc de Berg et. al. "Computational Geometry".

1 Max Flow

1.1 Disposition

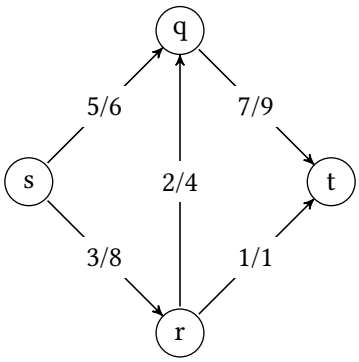
(1) Sketch presentation outline.

(2) Introduction to the topic.

(3) Max-flow example.

(4) Edmonds-Karp analysis.

Example:



1.2 Presentation

1. **Intro:** Flow networks and flow problems are problems where we want to transport material across edges of a graph, from a source node s to a sink node t. There are many important terms in this topic which I am going to skip, simply because I do not have the time to to through it all AND talk about Emonds-Karp. You can always ask, and hopefully my example will help with the understanding as well (draw example).

2. But, I do want to briefly define what flow f in such a graph is. It's a function which takes two nodes of the graph and results in a real value for the edge between the two nodes. Flow in a graph must satisfy capacity constraints and flow conservation, which intuitively makes sense if you look at the example (point to example graph). The value $|f|$ of a flow f is the net outflow from the source node

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s).$$

   In max-flow problems we want to find a flow with maximum possible value.

3. We can do this using the Ford-Fulkerson method. The method initializes a flow to 0, then continuously pick an augmenting path p in the residual network of G and augments the flow f along the path. When this can be done no more, since there are no augmenting paths, it returns the resulting flow f.

   So, there are some terms here that have not been explained. I want to briefly show what the residual graph is, and then show and explain an augmenting path. Also, augmenting a flow is done by $|f \uparrow f_p| = |f| + |f_p|$, which I can explain if you want, but at first I am going to skip that for the sake of time.

4. OK, so a residual network is made up of the same nodes, but each edge is determined by the flow of the original graph. All edges are strictly positive, so we remove any edges that might have capacity 0. An augmenting path is then a path from s to t in the residual graph, on which we can increase the flow. (Show path example in graph).

5. So, the Ford-Fulkerson method is correct, because of something called the min-cut max flow theorem, which states that if there are no augmenting paths in the residual graph, then f is a max-flow. It also states that any cut which separates the graph into S and $T = V - S$, such that $s \in S$ and $t \in T$, then this cut will have the a cost $c(S, T)$ equal to the value of the flow f. This involved defining the net flow across the cut and the cost of the cut, which I will skip. For now, notice that this theorem guarantees that the Ford-Fulkerson method is correct.

6. And so, let's talk about the Edmon-Karp algorithm. Edmond-Karp uses the Ford-Fulkerson method, where the chosen augmenting path in the residual graph is always the shortest path with respect to the number of edges in the path. I now want to analyze this algorithm and argue that its running time is $O(VE^2)$.

7. I am going to give the alternative proof which Christian uploaded to Absalon. So, we have a flow graph G with vertices V and edges E, source s and sink t, and a residual graph G_f. We are going to use something called forward edges, which are edges in the residual graph such that the shortest path distance from s increases by 1:

$$(u, v) \in E_f \ : \ \delta_f(s, v) = \delta_f(s, u) + 1$$

   These edges sort of divide the graph into levels, where each level i consists of nodes that have a shortest path from s of length i.

8. We also need a lemma, which basically says that during the iterations of the algorithm, flow is only pushed along forward edges in the original residual graph, and that the shortest path from s to t in the residual graph does not get shorter during the iterations.

9. I am going to write this properly, because I want to prove it: Let G_{f_0} be a residual network obtained at some point in the algorithm and let $f_1, \ldots, f_k$ be the sequence of flows obtained during the next $k \geq 0$ iterations. Let $d = \delta_{f_0}(s, t)$ and assume that $\delta_{f_i}(s, t) = d$ for $i \in [k-1]$. Then in each of the k iterations, flow is only pushed along edges that are forward edges in G_{f_0}. Furthermore, $\delta_{f_k}(s, t) \geq d$.

10. OK, so I want to prove this by induction over k. The base case, with $k = 0$ is easy, since we have not iterations and thus $G_{f_k} = G_{f_0}$.

11. For the induction case, assume that $k > 0$ and that the claim holds for $k - 1$. That is, that flow has only been pushed along forward edges so far. I now need to show that flow is actually also pushed along forward edges in the kth iteration.

12. So we know that Edmonds-Karp augments flow along the shortest path p from s to t, and since this path is in the residual graph for $k-1$, we know that all the edges of p are forward edges of G_{f_0}, and thus we must have that

$$(u, v) \in p \implies \delta_{f_0}(s, v) \leq \delta_{f_0}(s, u) + 1$$

since we can at most deviate 1 from the shortest path to u by going one level up; we could potentially move back toward s, by going one level down, or staying on the same level.

13. That means that either (u, v) is an edge of G_{f_0}, if we do not cancel flow in the kth iteration, OR it is the reversal of a forward edge of G_{f_0} in the case where we cancel flow. And since flow only runs along forward edges, the reversal must be of a forward edge.

14. We also know that the length of p is $\delta_{f_0}(s, t)$, which implies that actually, it must be the case that for every $(u, v) \in p$, we have that $\delta_{f_0}(s, v) = \delta_{f_0}(s, u) + 1$, since each edge in p can get you at most 1 close to the sink, and because there are exactly d edges in p, then all edges must get you precisely one closer if the total distance is d. And since this is the definition of a forward edge, then we have shown that all edges in p are in fact forward edges.

15. OK, so now we still need to show that the shortest path in the new residual graph is at least as long as d.

    Well, we have just argued that any such path p must have at least d edges, and since each edge carry weight 1, we have that $\delta_{f_k} \geq d$.

16. So that was the lemma which we are going to need now in the actual time bound. Namely, we will use it to show that the Edmonds-Karp algorithm runs in $O(VE^2)$ time.

17. Since we use breadth first search to find the shortest path in the residual network, we know that each iteration takes $O(V + E)$ time, which is actually $O(E)$ since we assume that the graph is connected. Thus we need to show that there are only $O(VE)$ iterations.

18. We now look at the largest sequence of k consecutive flows in the algorithm, where the shortest distance from s to t remains the same. And now we need the lemma, which we can use to infer that k is a most the number of edges in the original graph. Why? Because each iteration removes a forward edge, which there can be at most $|E|$ of. If no forward edge was removed, we could have pushed more flow along the chosen path. Once removed, you know that the forward edge does not reappear during the remaining of the k iterations,

since that would contradict the lemma, which we just saw meant that no flow is pushed along the reverse of a forward edge.

19. We also know that the shortest path distance after the final iteration is greater than or equal to $d + 1$, since it must be higher than or equal to d by the lemma, and different from d by the fact that we chose the largest sequence of iterations where d stayed the same.

20. Wrapping it up, we know that for every $|E|$ iterations, the distance in the residual graph is guaranteed to increase by at least 1. We also know that the distance can never exceed $|V| - 1$ because then you have visited all nodes in the graph. Thus, we go through all edges once for all nodes in the graph, which is $O(VE)$ time.

2 Linear Programming and Optimization

2.1 Disposition

(1) Sketch presentation outline.

(2) Introduction.

(3) Objective function, constraints, forms of LP's.

(4) Example.

(5) SIMPLEX algorithm for solving linear programs.

(6) Proof of weak duality.

Example:

$$\begin{aligned} \text{minimize } & -2x_1 + 3x_2 \\ \text{subject to } & x_1 + x_2 = 7 \\ & x_1 - 2x_2 \leq 4 \\ & x_1 \geq 0. \end{aligned}$$

Convert to standard form, then convert to slack form. Then simplex.

2.2 Presentation

1. Introduction: We care about linear programs because many real life problems can be modelled as linear programs, and so this topic is directly useful in many practical settings.
2. A general linear program consists of an objective function that describes some relation between the variables of the problem, which we want to either maximize or minimize; it also consists of linear constraints on the variables. Usually we write linear programs in standard form, or in slack form, which are simply specific ways to represent a linear problem.
3. In **standard form** we are given n real numbers $c_1, c_2, \ldots, c_n$ and m real numbers $b_1, b_2, \ldots, b_m$ and nm real numbers a_{ij} for $i \in [m]$ and $j \in [n]$. We wish to assign values to n variables $x_1, x_2, \ldots, x_n$ that

$$\begin{aligned} \text{maximize} \quad & \sum_{j=1}^{n} c_j x_j & \\ \text{subject to} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i & \text{for } i \in [m] \\ & x_j \geq 0 & \text{for } j \in [n]. \end{aligned}$$

   Namely, we always maximize the objective function, and there are non-negativity constraints on all the n variables. Also, every inequality must be $\leq$ and not $=$ or $\geq$. (convert example into standard form).
4. Similarly, in slack form, which we use for the ensuing SIMPLEX algorithm, all inequalities that are not non-negativity constraints must be strict equalities. We do this by introducing slack variables (convert example into slack form).
5. Now, the simplex algorithm works by continuously changing the basic solution by pivoting variables to and from the basic variables. Pick a variables in the objective function which positively increases the value, and pivot it with the basic variable that bottlenecks how much the non-basic variable can be increased. (Show simplex on example).
6. The SIMPLEX algorithm is more complex than this, since we have many special cases, such as when the feasible region of the program is unbounded, or when the initial basic solution is not feasible. So, there are more to the algorithm than what I have just shown. You also need to argue why the procedure actually terminates and why this actually computes the optimal solution to the problem.
7. For the latter, we have something called the dual of a linear program, which we can use to show that the SIMPLEX algorithm actually computes the optimal solution to the problem. I just want to talk about the weak duality theorem, which states that the solution to the DUAL of a linear program (PRIMAL) is always an upper bound to the solution to the PRIMAL.
8. The DUAL looks as follows:

$$\begin{aligned} \text{minimize} \quad & \sum_{i=1}^{m} b_i y_i & \\ \text{subject to} \quad & \sum_{i=1}^{m} a_{ij} y_i \geq c_j & \text{for } j \in [n] \\ & y_i \geq 0 & \text{for } i \in [m]. \end{aligned}$$

9. So, what the weak duality theorem says is that it is always the case that

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{i=1}^{m} b_i \bar{y}_i,$$

which is the case because

$$\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j &\leq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j && \text{by the inequality of the dual constraints,} \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i && \text{since this is simply a reordering of the summation terms} \\
&\leq \sum_{i=1}^{m} b_i \bar{y}_i && \text{by the primal inequality constraints.}
\end{aligned}$$

3 Randomized Algorithms

3.1 Disposition

(1) Sketch presentation outline.

(2) Motivation: For many applications, a randomized algorithm is the simplest algorithm available, or the fastest, or both.

(3) Monte Carlo and Las Vegas.

(4) Example: **RandQS**.

Call on example $3, 5, 2, 1, 4$. With first pivot 2, then 4. We have $\pi = 2, 1, 4, 3, 5$.

(5) Example: **Min-Cut** algorithm.

3.2 Comprehensive Notes

3.2.1 RandQS Algorithm

The following algorithm **RandQS** is an example of a *randomized algorithm*, which sorts in input set of numbers. It is a Las Vegas algorithm, meaning that the output is always correct, but the runtime may vary.

Algorithm 1: RandQS

Data: A set of numbers S.

Result: The elements of S sorted in increasing order.

1 Choose an element y uniformly at random from S: every element in S has equal probability of being chosen.

2 By comparing each element of S with y, determine the set S_1 of elements smaller than y and the set S_2 of elements larger than y.

3 Recursively sort S_1 and S_2. Output the sorted version of S_1, followed by y, and then the sorted version of S_2.

Analysis of expected running time for **RandQS**:

i. As is usual for sorting algorithms, we measure the running time in terms of the number of comparisons it performs. In particular, our goal is to analyze the *expected* number of comparisons. Note that all the comparisons are done in step 2, in which we compare a randomly chosen partitioning element to the remaining elements.

ii. for $1 \leq i \leq n$, let $S_{(i)}$ denote the element of *rank* i (the ith smallest element) in the set S. Thus, $S_{(1)}$ is the smallest element and $S_{(n)}$ is the largest.

iii. Define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and $S_{(j)}$ are compared in an execution and the value 0 otherwise. Thus, the total number of comparisons is $\sum_{i=1}^{n} \sum_{j>i} X_{ij}$.

iv. We are interested in the expected number of comparisons, which is clearly

$$\mathbb{E}\left[\sum_{i=1}^{n} \sum_{j>i} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i} \mathbb{E}\left[X_{ij}\right]$$

by linearity of expectation.

v. Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared. Since X_{ij} is an indicator variable, we have $\mathbb{E}\left[X_{ij}\right] = p_{ij}$. Next, we need to determine what p_{ij} is.

vi. View the execution of **RandQS** as a binary tree T, each node of which is labeled with a distinct element of S. the root of the tree is labeled with the element y chosen in step 1. The left sub-tree of y is S_1 and the right sub-tree is S_2. The structures of the two sub-trees are determined recursively by the executions of **RandQS** ans S_1 and S_2.

vii. The root y is compared to the elements in the sub-trees, but no comparison is performed between an element of S_1 and an element of S_2. Thus, there is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if one of these elements is an ancestor of the other.

viii. The in-order traversal of T is the sorted order of S (and also the output of the algorithm). For this analysis we are interested in the level-order traversal. This is the permutation π obtained by visiting the nodes of T in increasing order of the level numbers, and in a left-to-right order within each level; recall that the ith level of the tree is the set of all nodes at distance exactly i from the root.

ix. Elaboration: π will be first the root of the entire tree T, then the root of the left sub-tree, followed by the root of the right sub-tree. Then the root of the left sub-tree of the left sub-tree, and so on.

x. To compute p_{ij} we make two observations:

1. $S_{(i)}$ and $S_{(j)}$ are compared if and only if $S_{(i)}$ or $S_{(j)}$ occurs earlier in the permutation π than any element $S_{(\ell)}$ such that $i < \ell < j$.

   To see this, let $S_{(k)}$ be the earliest in π from among all elements of rank between i and j.

   If $k \notin \{i, j\}$, then $S_{(i)}$ will belong to the left sub-tree of $S_{(k)}$ while $S_{(j)}$ will belong to the right sub-tree of $S_{(k)}$, implying that there is no comparison between $S_{(i)}$ and $S_{(j)}$.

   If $k \in \{i, j\}$, there is an ancestor-descendant relationship between $S_{(i)}$ and $S_{(j)}$, implying that the two elements are compared by **RandQS**.

2. Any of the elements $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$ is equally likely to be the first of these elements to be chosen as a partitioning element, and hence to appear first in π. Thus, the probability that this first element is either $S_{(i)}$ or $S_{(j)}$ is exactly $\frac{2}{j-i+1}$.

xi. Thus $p_{ij} = \frac{2}{j-i+1}$. The expected number of comparisons is then given by

$$\begin{aligned} \sum_{i=1}^{n} \sum_{j>i} p_{ij} &= \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j-i+1} && \text{substituting } p_{ij} = \frac{2}{j-i+1}, \\ &\le \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{2}{k} \\ &\le 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k}. \end{aligned}$$

It follows that the expected number of comparisons is bounded above by $2nH_n$, where $H_n = \sum_{i=1}^{n} 1/i$. We have that $H_n \sim \ln n + \Theta(1)$, so that the expected running time of **RandQS** is $O(n \log n)$. This expected running time holds for every input.

3.2.2 Min-Cut Algorithm

Let G be a connected, undirected multigraph with n vertices. A *multigraph* may contain multiple edges between any pair of vertices. A *cut* in G is a set of edges whose removal results in G being broken into two or more components. A *min-cut* is a cut of minimum cardinality.

Algorithm 2: MIN-CUT

Data: A graph G with n vertices.

Result: A candidate min-cut, i.e. a set of edges of G.

1 **Repeat:** Pick and edge uniformly at random and merge the two vertices at its end-points. If as a result there are several edges between some pairs of newly formed vertices, retain them all. Edges between vertices that are merged are removed, so that there are never any self-loops. This process is called a *contraction* of the edge.

2 With each contraction, the number of vertices of G decreases by 1. The crucial observation is that an edge contraction does not reduce the min-cut size in G. This is because every cut in the graph at any intermediate stage is a cut in the original graph.

3 The algorithm continues the contraction process until only two vertices remain; at this point, the set of edges between these two vertices is a cut in G and is output as a candidate min-cut.

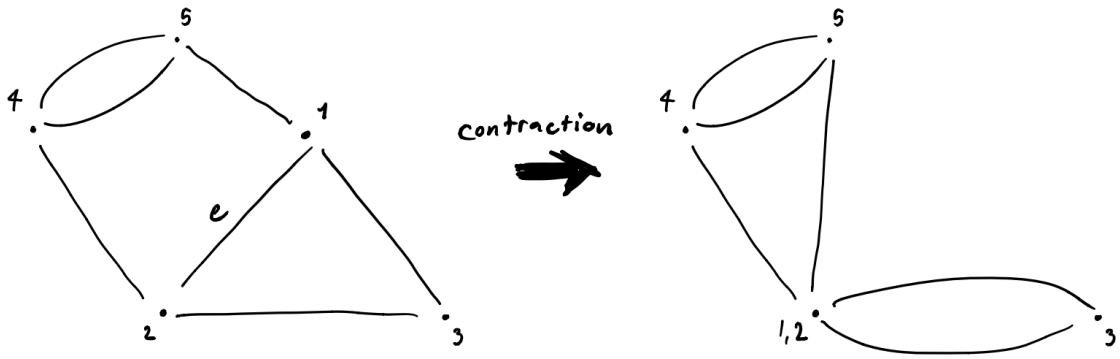


Figure 1: Contraction of edge e.

Analysis of failure probability:

i. Let k be the min-cut size. We fix our attention on a particular min-cut C with k edges. Clearly G has at least $kn/2$ edges; otherwise there would be a vertex of degree less than k, and its incident edges would be a min-cut of size less than k.

ii. We will bound from below the probability that no edge of C is ever contracted during an execution of the algorithm, so that the edges surviving till the end are exactly the edges in C.

iii. Let $\mathcal{E}_i$ denote the event of *not* picking an edge of C at the ith step, for $1 \leq i \leq n-2$. The probability that the edge randomly chosen in the step is in C is a most $\frac{k}{\left(\frac{nk}{2}\right)} = \frac{2}{n}$, so that $\Pr[\mathcal{E}_1] \geq 1 - \frac{2}{n}$.

iv. Assuming that $\mathcal{E}_1$ occurs, during the second step there are at least $\frac{2(n-1)}{2}$ edges, so the probability of picking an edge in C is at most $\frac{2}{(n-1)}$, so that $\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] \geq 1 - \frac{2}{(n-1)}$.

v. At the ith step, the number of remaining vertices is $n - i + 1$. The size of the min-cut is still at least k, sp the graph has at least $\frac{k(n-i+1)}{2}$ edges remaining at this step. Thus $\Pr[\mathcal{E}_i \mid \cap_{j=1}^{i-1}\mathcal{E}_j] \geq 1 - \frac{2}{(n-i+1)}$

vi. Looking at the entire process, what is then the probability that no edge of C is ever picked? It is

$$\Pr[\cap_{i=1}^{n-2}\mathcal{E}_i] \geq \prod_{i=1}^{n-2}\left(1 - \frac{2}{n-i+1}\right) = \frac{2}{n(n-1)}.$$

vii. The probability of discovering a particular min-cut (which may be unique) is larger than $\frac{2}{n^2}$. Thus our algorithm may err in declaring the cut it outputs to be a min-cut.

viii. Suppose we repeat the algorithm $\frac{n^2}{2}$ times, making independent random choices each time. Then the probability that a min-cut is *not* found in any of the $\frac{n^2}{2}$ attempts is at most

$$\left(1 - \frac{2}{n^2}\right)^{\frac{n^2}{2}} < \frac{1}{e}.$$

By this process of repetition, we have managed to reduce the probability of failure from $1 - \frac{2}{n^2}$ to a more respectable $\frac{1}{e}$.

ix. Further repetitions of the algorithm will make the failure probability arbitrarily small, the only consideration being that this increases the overall running time.

3.2.3 Las Vegas and Monte Carlo

The algorithm **RandSQ** is a **Las Vegas** algorithm, since it always outputs the correct answer.

The algorithm **Min-Cut** is a **Monte Carlo** algorithm, since it sometimes produces a solution that is incorrect.

For decisions problems there are two kinds of Monte Carlo algorithms: those with *one-sided error* and those with *two-sided error*. A Monte Carlo algorithm is said to have two-sided error if there is a non-zero probability that it errs when it outputs either YES or NO. It is said to have one-sided error if the probability that it errs is zero for at least one of the possible outputs.

3.3 Presentation

1. **Motivation:** Randomized algorithms are algorithms that algorithms that utilizes randomness in their procedure. They are often the simplest or the fastest alternative in practical settings, and some problems are practically only solvable using randomization.

   There are two types of randomized algorithms. One is the Las Vegas algorithm, which always returns the correct answer, and we study the runtime, which might vary.

   The other type is the Monte Carlo algorithm, which is fast, but not guaranteed to be correct in its answer.

   I will give two examples, one of each type of randomized algorithm.

2. **Randomized QuickSort:** So the first algorithm is randomized quick sort, which is used for sorting a set of elements. It is similar to quick sort; the only difference is that the pivot value is chosen at random.

   Let's go through a short example: $[3, 5, 2, 1, 4]$. We first pivot on 2 and then 4. The resulting decision tree we call T, and the output of the randQS is the inorder traversal. The level-order traversal, which we are going to use for the runtime analysis is $\pi = 2, 1, 4, 3, 5$.

3. OK, so what is the expected runtime? We measure this by the number of comparisons. To do this, we use an indicator variable X_{ij} to denote whether element i and j are compared in the process. The total number of comparisons is then

   $$\sum_{i=1}^{n} \sum_{j>i}^{n} X_{ij}$$

   and the expected number of comparisons is then

   $$\mathbb{E}\left[\sum_{i=1}^{n} \sum_{j>i}^{n} X_{ij}\right] = \sum_{i=1}^{n} \sum_{j>i}^{n} \mathbb{E}\left[X_{ij}\right]$$

   by linearity of expectation. And since X_{ij} is an indication variable, the expectation of it, is equal to the probability p_{ij} that i and j are compared in the process. So, we need to figure out what this probability is.

4. OK, so to do that, observe that it is only the chosen pivot which is compared to other elements. And looking at our level-order walk, we can see that i and j are only compared if $i < l < j$ is not before any of the two in the permutation, since if there were such an element l, then i and j would end up in different subtrees, and thus are not compared.

5. So this means that i and j are only ever compared if one of them is chosen first from all the elements in the range from i to j. There are $j - i + 1$ such elements, and we choose the pivot uniformly at random, so p_{ij} is then $\frac{2}{j-i+1}$.

6. Now, we can then use this in our calculation:

   $$\sum_{i=1}^{n} \sum_{j>i}^{n} \frac{2}{j - i + 1}$$

   which is the same as

   $$\sum_{i=1}^{n} \left(\frac{2}{2} + \frac{2}{3} + \ldots + \frac{2}{n - i + 1}\right)$$

and this is smaller than or equal to taking the second sum over the entire range of n:

$$2\sum_{i=1}^{n}\sum_{k=1}^{n}\frac{1}{k}.$$

here we have $2n$ of the nth harmonic number, which is $O(\ln n)$, meaning that the expected number of comparisons are $O(n \log n)$.

7. **Min-Cut Algorithm:**

4 Hashing

4.1 Disposition

(1) Sketch presentation outline.

(2) Motivation: Why do we care about hashing.

(3) Definitions: Universal and strongly universal hashing.

(4) Example: Hash table with chaining. Constant expected search time.

(5) Show that `Multiply-shift` is 2-universal.

4.2 Comprehensive Notes (Mostly HSHIS Paper)

Usually the idea behind hashing is that the universe U of keys that we wish to map is much too large to contain. Hence we map it to a smaller range $m = \{0, \dots, m-1\}$ of hash values.
Definition (hash function)**.** *A hash function $h : U \longrightarrow [m]$ is a random variable in the class of all functions $U \longrightarrow [m]$, that is, it consists of a random variable $h(x)$ for each $x \in U$.*

We care about three things: (1) Space, which is the size of the random seed that is necessary to calculate $h(x)$ given x, (2) Speed, which is the time it takes to calculate $h(x)$ given x, and (3) the properties of the random variable. Some random variables have desirable properties such as universal hash functions, or strongly universal hash functions.
Definition (universal and c-universal hash function)**.** *Let $h : U \longrightarrow [m]$ by a random hash function from a key universe U to a set of hash values $[m] = \{0, \dots, m-1\}$. Think of h as a random variable following some distribution over functions $U \longrightarrow [m]$. Such a hash function h is **universal** if for any given distinct keys $x, y \in U$, when h is picked at random independently of x and y, we have a **low collision probability**:*

$$\Pr_h[h(x) = h(y)] \leq \frac{1}{m}.$$

*Also, h is called c-**universal** if, for some $c = O(1)$, we have:*

$$\Pr_h[h(x) = h(y)] \leq \frac{c}{m}.$$

Example 1. *One of the most classic applications of universal hashing is hash tables with chaining. We wish to store a set $S \subseteq U$ of keys such that we can expect to find a key from S in constant time. Let $|S| = n$ and $m \geq n$. We pick a universal hash function $h : U \longrightarrow [m]$, and then create an array L of m lists/chains so that for $i \in [m]$, $L[i]$ is the list of keys that hash to i.*

To decide if a key $x \in U$ is in S, we only have to check if x is in the list $L[h(x)]$. This takes time proportional to $1 + |L[h(x)]|$, since we, in any case, spent 1 on look-up, and then subsequently the cost of going through $L[h(x)]$.

Assume that $x \notin S$ and that h is universal. Let $I(y)$ be an indicator variable which is 1 if $h(x) = h(y)$ and 0 otherwise. Then the expected number of elements in $L[h(x)]$ is:

$$\mathbb{E}\left[|L[h(x)]|\right] = \mathbb{E}\left[\sum_{y \in S} I(y)\right] = \sum_{y \in S} \mathbb{E}\left[I(y)\right] = \sum_{y \in S} \Pr[h(x) = h(y)] = \frac{n}{m} \leq 1.$$

Example 2. *A different application is that of assigning a unique signature $s(x)$ to each key. Thus we want $s(x) \neq s(y)$ for all distinct key $x, y \in S$. To get this, we pick a universal hash function $s : U \longrightarrow n^3]$. The probability of error (collision) is calculated as*

$$\Pr[\exists \{x, y\} \subseteq S : s(x) = s(y)] \leq \sum_{\{x,y\} \subseteq S} \Pr[s(x) = s(y)] = \binom{n}{2} n^{-3} < \frac{1}{2n}$$

The idea of signatures is particularly relevant when the keys are large, e.g., a key could be a whole text document.

This idea could be used in connection with hash tables, letting the list $L[i]$ store the signatures $s(x)$ of the keys that hash to i, that is, $L[i] = \{s(x) \mid x \in X, h(x) = i\}$. To check if x is in the table, we then check whether $s(x)$ is in $L[h(x)]$.

4.2.1 Hashing Schemes

Definition (Multiply-mod-prime)**.** *The classical universal hash function is based on a prime number $p \geq u$. We pick a uniformly random number $a \in [p]_+ = \{1, \ldots, p-1\}$ and $b \in [p] = \{0, \ldots, p-1\}$, and define $h_{a,b} : [u] \longrightarrow [m]$ by*

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m.$$

This hash function has a collision probability strictly lower than $1/m$. The proof is not curriculum.

Definition (Multiply-Shift)**.** *`Multiply-shift` is a practical universal hashing scheme. It generally addresses hashing from w-bit integers to ℓ-bit integers. Pick a uniformly random odd w-bit integer a, and then compute $h_a : [2^w] \longrightarrow [2^\ell]$ as*

$$h_a(x) = \left\lfloor \frac{(ax \mod 2^w)}{2^{w-\ell}} \right\rfloor$$

This scheme gains an order of magnitude in speed over Multiply-mod-prime, exploiting operations that are fast on standard computers. With $w = 64$, the following C-code is a simple, sufficient implementation:

```
#include <stdint.h>
uint64_t hash(uint64_t x; uint64_t l; uint64_t a) {
    return (a * x) >> (64 - l);
}
```

Multiply-shift, is 2-universal, meaning that for $x \neq y$ and a random odd w-bit integer a (with $\Pr = \Pr_{\text{odd } a \in [2^w]}$ and $h = h_a$),

$$\Pr[h(x) = h(y)] \leq \frac{2}{2^\ell} = \frac{2}{m}$$

Proof. Think of the bits of a number as indexed with bit 0 the least significant bit. The scheme is simply extracting bits $w - \ell, \ldots, w - 1$ from the product ax. That is, it extracts the ℓ most significant bits of $ax \mod 2^w$, ensuring that h_a is indeed $[2^w] \longrightarrow [2^\ell]$.

We have $h_a(x) = h_a(y)$ if and only if ax and $ay = ax + a(y - x)$ agree on bits $w - \ell, \ldots, w - 1$. This match requires that bits $w - \ell, \ldots, w - 1$ of $a(y - x)$ are either all 0s or 1s. More precisely, if we get no carry from bits $0, \ldots, w - \ell$ when we add $a(y - x)$ to ax, then $h_a(x) = h_a(y)$ exactly when bits $w - \ell, \ldots, w - 1$ of $a(y - x)$ are all 0s. Similarly, if we get a carry, then $h_a(x) = h_a(y)$ when the same bits are all 1s, since the carry is then carried all the way through the bits, leaving all 0s behind. It is thus sufficient to show that the probability of the bits being all 0 or 1s is at most $\frac{2}{2^\ell}$.

To do this, we use the fact that any odd number z is relatively prime to any power of two:

$$\text{if } \alpha \text{ is odd and } \beta \in [2^q]_+ \text{ then } \alpha\beta \not\equiv 0 \pmod{2^q}. \tag{1}$$

Define b such that $a = 1+2b$. Then b is uniformly distributed in $[2^{w-1}]$, since a is chosen uniformly at random. Moreover, define z to be the odd number satisfying $(y - x) = z2^i$. Then

$$a(y - x) = (1 + 2b)(y - x) = z2^i + bz2^{i+1}.$$

Next we show that $bz \mod 2^{w-1}$ is uniformly distributed in $[2^{w-1}]$. There is a 1 to 1 correspondence between the $b \in [2^{w-1}]$ and the products $bz \mod 2^{w-1}$; if not, then there would be a b' such that $b'z \equiv bz \pmod{2^{w-1}} \iff z(b - b') \equiv 0 \pmod{2^{w-1}}$. This is a contradiction to fact (1),

since z is odd. But then the uniform distribution on b implies that $bz \mod 2^{w-1}$ is uniformly distributed.

We can now say that $a(y-x)$, as specified above, has 0 in bits $0, \ldots, i-1$, because of shifting with at least 2^i; it has 1 at index i, since z is odd, and thus we must have that $z2^i$ has a 1 at bit index i; and finally, a uniform distribution on bits $i+1, \ldots, i+w-1$, because bz is uniformly distributed and shifted to index 2^{i+1}.

Concluding the proof, if $i \geq w-\ell$, then $h_a(x) \neq h_a(y)$, since ax and ay are always different in bit i; however, if $i < w-\ell$, then, because of carries, we could have $h_a(x) = h_a(y)$ if bits $w-\ell, \ldots, w-1$ of $a(y-x)$ are all either 0s or 1s. Because of the uniform distribution, either event happens with probability $\frac{1}{2^\ell}$, for a combined probability bounded by $\frac{2}{2^\ell}$, which is what we needed to prove. □

Definition (Strong universality)**.** *For $h : [u] \longrightarrow [m]$, we consider pair-wise events of the form that for given distinct keys $x, y \in [u]$ and possibly non-distinct hash values $q, r \in [m]$ we have $h(x) = q$ and $h(y) = r$. We say a random hash function $h : [u] \longrightarrow [m]$ is **strongly universal** if the probability of every pair-wise event is $1/m^2$. We note that if h is strongly universal, it is also universal since*

$$\Pr[h(x) = h(y)] = \sum_{q \in [m]} \Pr[h(x) = q \wedge h(y) = q] = \frac{m}{m^2} = \frac{1}{m}.$$

An equivalent definition of strong universality is that each key is hashed uniformly into $[m]$, and that every two distinct keys are hashed independently.

Definition (Strongly c-universal hash function)**.** *We say a random hash function $h : U \longrightarrow [m]$ is strongly c-universal if*

1. *h is c-uniform, meaning for every $x \in U$ and for every hash value $q \in [m]$, we have $\Pr[h(x) = q] \leq c/m$, and*
2. *every pair of distinct keys hash independently.*

4.3 Presentation

1. **Motivation:** Hashing is used for handling data of large size by mapping it into data of fixed size. Usually, the idea behind hashing is that the universe U of keys that we wish to map is much too large to contain in memory. Hence we map it to a smaller range $m = \{0, 1, \dots, m-1\}$ of hash values.

   So, a hash function h is a function from U to $[m]$, where $h(x)$ is a random variable.

   One disadvantage is that hash values are not unique, and thus we might have collisions between the hash values of keys from U.

   Because of this, some hash functions might be preferable to others, since some may have properties which guarantees a low collision probability.

2. **Universal hash functions:** One such example is the universal hash functions. The definition is, that for any to distinct keys x and y from U, the probability of collision $\Pr[h(x) = h(y)]$ is less than or equal to $\frac{1}{m}$.

… Incomplete.

5 van Emde Boas Trees

5.1 Disposition

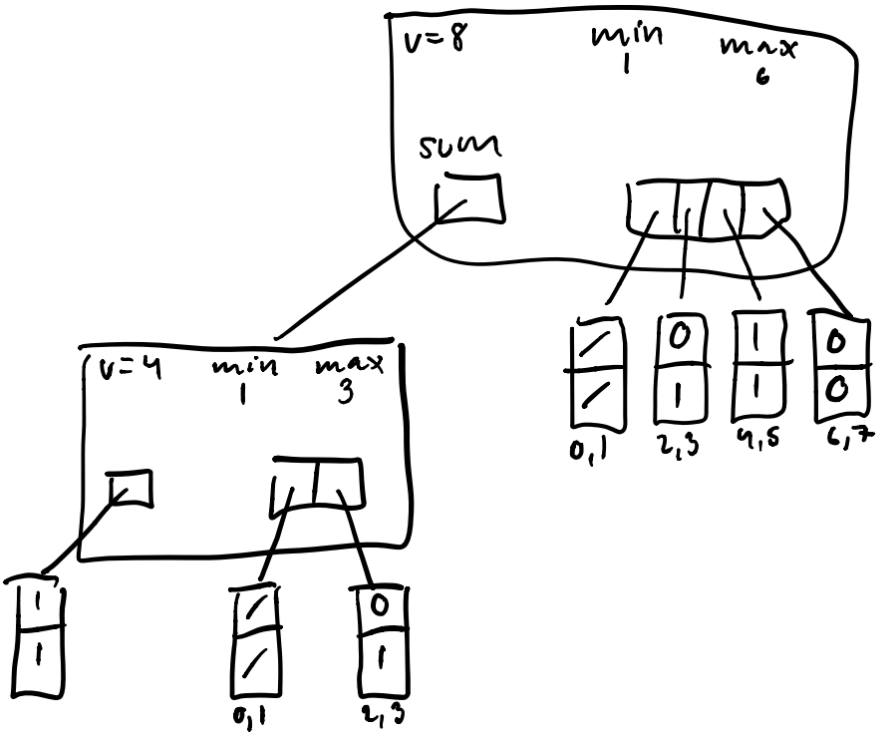
(1) Sketch presentation outline.

(2) Motivation: Why are van Emde Boas Trees interesting.

(3) What is it.

(4) Running time analysis.

Example:



It stores $\{1, 2, 3, 5, 6\}$. Insert 7. Insert 0. Find successor of 3.

5.2 Comprehensive Notes

Knowing the range of a set of keys being sorted, we can exploit this additional information to construct sorting algorithms that run asymptotically faster then the $O(n \log n)$ bound that we are used to. In a similar fashion, the **van Emde Boas Tree** data structure exploits this information to construct a priority-queue structure with operations supported in $o(\lg n)$ time; van Emde Boas Trees support the priority-queue operations in $O(\lg \lg u)$ worst-case time. The hitch is that the keys must be integers in the range 0 to $u - 1$, with no duplicates allowed.

The operations supported are: Search, Insert, Delete, Minimum, Maximum, Successor and Predecessor. All in $O(\lg \lg u)$ time.

The analysis is based on the recurrence $T(u) = T\left(\sqrt{u}\right) + O(1)$. Let $m = \lg u$, so that $u = 2^{\lg m}$ and we have

$$T(2^m) = T\left(2^{\frac{m}{2}}\right) + O(1).$$

Rename $S(m) = T(2^m)$, giving the recurrence

$$S(m) = S\left(\frac{m}{2}\right) + O(1).$$

Using case 2 of the master method, this recurrence has the solution $S(m) = O(\lg m)$. Substituting $S(m)$ with $T(u)$, we have that $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u)$.

Because of this, we will look for a data structure with this recurrence. That is, when calling an operation on the data structure, this operation will traverse the data structure such that any recursive call shrinks by a factor of $\sqrt{u}$ and such that it will spend a constant amount of time at each level before recursing to the level below. Then the above recursion will describe the running time of the operation.

We will allow the universe size u to be any exact power of 2, and when $\sqrt{u}$ is not an integer, then we will divide the $\lg u$ bits of a number into the most significant $\left\lceil \frac{(\lg u)}{2} \right\rceil$ bits and the least significant $\left\lfloor \frac{(\lg u)}{2} \right\rfloor$ bits. For convenience, we denote the resulting numbers as the upper square root of u, $\uparrow\sqrt{u}$ and the lower square root, $\downarrow\sqrt{u}$ of u, so that $u = \uparrow\sqrt{u} \cdot \downarrow\sqrt{u}$, and, when u is an even power of 2, we have $\downarrow\sqrt{u} = \uparrow\sqrt{u} = \sqrt{u}$.

We define our helpful functions as follows:

$$\begin{aligned} \text{high}(x) &= \left\lfloor \frac{x}{\downarrow\sqrt{u}} \right\rfloor, \\ \text{low}(x) &= x \mod \downarrow\sqrt{u}, \\ \text{index}(x, y) &= x\downarrow\sqrt{u} + y. \end{aligned}$$

Definition (van Emde Boas tree)**.** *The **van Emde Boas tree**, denoted **vEB tree**, is a recursive data structure on a universe of keys, whose size u is any exact power of 2. Each $vEB(u)$ contains the universe size u, elements* min *and* max*, a pointer **summary** to a $vEB(\uparrow\sqrt{u})$ tree, and an array cluster $[0 \ldots \uparrow\sqrt{u} - 1]$ of $\uparrow\sqrt{u}$ pointers to $vEB(\downarrow\sqrt{u})$ trees.*

min *stores the minimum element in the vEB tree, and* max *stores the maximum element in the vEB tree. The element stored in* min *does not appear in any of the recursive vEB trees, which means that the tree consists of all the elements stored recursively AND the minimum element. The maximum element IS stored in the recursive structure.*

This way, we obtain several desirable properties about the operations on the data structure:

1. The Minimum and Maximum operations do not need to recurse, for they can just return the value of min or max.
2. The Successor operation can avoid making a recursive call to determine whether the successor of a value x lies within high(x). That is because x's successor lies within its cluster if and only if x is strictly less than the max attribute of its cluster. A symmetric argument hold for Predecessor and min.
3. We can tell if an vEb tree contains 0, 1, or at least 2 elements in constant time. This will help Insert and Delete operations.
4. If we know that a vEB tree is empty, we can insert into it in constant time, by only updating its min and max properties. Similarly, if the tree contains exactly 1 element, we can delete in constant time. These properties will allow us to cut short the chain of recursive calls.

The recursive procedures that implement the vEB-tree operations will all have running times characterized by the recurrence

$$T(n) \le T(\sqrt[\uparrow]{u}) + O(1).$$

Letting $m = \lg u$ we rewrite as

$$T(2^m) \le T\left(2^{\lceil m/2 \rceil}\right) + O(1).$$

Noting that $\left\lceil \frac{m}{2} \right\rceil \le \frac{2m}{3}$ for all $m \ge 2$, we have

$$T(2^m) \le T\left(2^{2m/3}\right) + O(1).$$

Letting $S(m) = T(2^m)$, we rewrite this last recurrence as

$$S(m) \le S\left(\frac{2m}{3}\right) + O(1),$$

which, by case 2 of the master method, has the solution $S(m) = O(\lg m)$. Thus we have $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u)$.

5.3 Presentation

1. Hello, I want to talk about the topic of van Emde Boas trees. First I just want to give you the outline of my presentation. I want to briefly give some motivation for why the data structure is interesting and why we should care about it, basically. Then, I am going to give you an idea of how we can analyse the running time of the operations, by using a recurrence relation. After than, I will probably use the remaining time on showing you what the tree looks like by an example, and running some operations on it. I do not think that I can include more than that into my 15 minutes.

2. **Motivation:** OK, so van Emde Boas trees are interesting mainly because of the bound on the running time on the operations on the structure. Similarly to sorting algorithms such as counting sort, the structure utilizes the additional knowledge of a given range of the input. In sorting, this allows us to sort in linear time instead of $n \log n$ time, while in dynamic sets, which the vEB is, it allows us to make basic operations such as predecessor, min, insert and so on, in $\lg \lg u$ size, where u is the size of the element universe.

3. there are some restrictions however, namely that we assume that the universe size is a power of 2, and that there are no duplicate elements.

4. Before I show you what the van Emde Boas tree looks like, I want to prepare you for the analysis of the runtime on operations. We are going to use a recurrence relation to analyse the operations, which is:

$$T(u) = T(\sqrt[\uparrow]{u}) + O(1),$$

   where $\sqrt[\uparrow]{u}$ is $2^{\left\lceil \frac{\lg u}{2} \right\rceil}$. If we can somehow show that the runtime of the operations follow this recurrence, then we are able to show that the runtime is $O(\lg \lg u)$, in the following way: Let $m = \lg u$, such that $T(u) = T(2^m) \leq T(2^{\lceil \frac{m}{2} \rceil}) + O(1)$, which for $m \geq 2$ is always smaller than or equal to $T(2^{2m/3}) + O(1)$. In one line:

$$T(u) = T(2^m) \leq T(2^{\lceil \frac{m}{2} \rceil}) + O(1) \leq T(2^{2m/3}) + O(1)$$

   Let $S(m) = T(2^m)$, such that $S(m) \leq S(2m/3) + O(1)$, which by case 2 of the master method theorem has a solution $S(m) = O(\lg m)$. Replacing, we get that $T(u) = T(2^m) = S(m) = O(\lg m) = O(\lg \lg u)$, which is the running time that we want our operations to have.

   Thus, if the running time of our operations can be described with this recurrence, then the running time is $O(\lg \lg u)$.

5. OK, so let us look at an example of a vEB. (Show example).

6 NP Completeness

6.1 Disposition

(1) Outline of presentation. Hard problems are problems that cannot be solved in polynomial time.

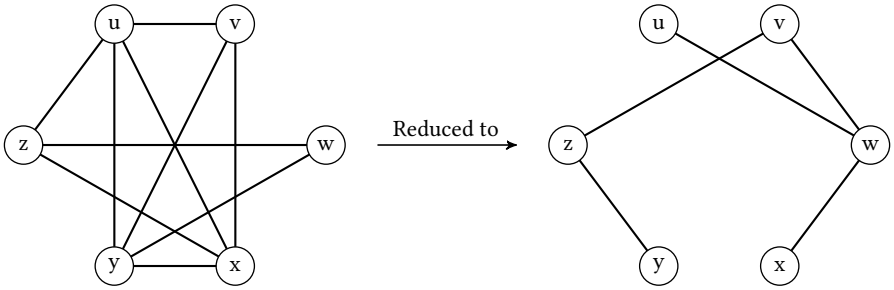
(2) Definition of verification and languages. Encoding.

(3) Definition of NP, and reduction.

(4) Definition of NP-completeness.

(5) Example1: Show NP-completeness for `VERTEX-COVER`.

(6) Example2: Show NP-completeness for `TSP`.



6.2 Comprehensive Notes (Mostly CW lecture slides)

Definition (Problem)**.** *Consider a set I of **instances** and a set S of **solutions**. An abstract **problem** is a binary relation between I and S, i.e., a subset of $I \times S$.*

E.g. for SHORTEST-PATH, an instance is a triple $\langle G, s, t\rangle$, where G is the graph, s is the source node and t is the sink node. A solution is a sequence of vertices forming a shortest s-to-t path.
Definition (Decision Problems)**.** *"Yes" or "No" problems, hence $S = \{0, 1\}$. E.g. PATH$(\langle G, u, v, k\rangle) = 1$ if there is a u-to-v path in G with at most k edges. Regard a decision problem as a mapping from instances to $S = \{0, 1\}$. Instances with solution 1 are called **yes**-instances. Guess what instances with solution 0 are called..*

Optimization problems can usually be turned into decision problems. That is, if you want to find an optimal path of min length, then iterate over the decision problem starting with the smallest k possible, increasing k by 1 for each iteration, and then stop when the answer to the decision problem is 1.
Definition (Polynomial-time solvable problems)**.** *We assume that instances of a problem are encoded as binary strings. An algorithm **solves** a problem in time $O(T(n))$ if for any instance of length n, the algorithm return a solution (0 or 1) in time $O(T(n))$. if $T(n) = O(n^k)$ for some constant k, the problem is **polynomial-time solvable**.*

we use $\langle x\rangle$ to refer to a chosen encoding if an instance x of a problem. Encoding are always binary string in our setting.
Definition (Languages)**.** *An **alphabet** is a finite set Σ og symbols. A **language** L over an alphabet Σ is a set of strings of symbols from Σ. Example: $\Sigma = \{a, b, c\}$ and $L = \{a, ba, cab, bbac, ...\}$. We also allow an empty string and denote it by ϵ. The empty language is denoted $\emptyset$ and it does not contain ϵ. Σ^* denotes the language of all strings (including ϵ). Any language L over Σ is a subset of Σ^*.*

Instances of decision problems are encoded as binary strings, and a decision problem itself can be regarded as a mapping $Q(x)$ from instances to $\Sigma = \{0, 1\}$. Q can be specified by the binary strings that encode yes-instances of the problem. Thus we can view Q as a language L:

$$L = \{x \in \Sigma^* \mid Q(x) = 1\}.$$

Definition (Accepting algorithm, deciding algorithm)**.** *Let A be an algorithm for a decision problem and denote by $A(x) \in \{0, 1\}$ its output given input x. Then A **accepts** a string x if $A(x) = 1$ and **rejects** a string x if $A(x) = 0$. The language accepted by A is:*

$$L = \{x \in \{0, 1\}^* \mid A(x) = 1\}.$$

*Suppose in addition that all string not in L are rejected by A, i.e. $A(x) = 0 \forall x \in \{0, 1\}^* \setminus L$. Then we say that L is **decided** by A. Deciding a language is stronger than accepting it.*

We say that a language L is accepted by an algorithm A in polynomial time if A accepts L and runs in polynomial time on strings from L. Similarly, L is decided by A in polynomial time if A decides L and runs in polynomial time on all strings.

Informally, a **complexity class** is a set of languages, membership in which is determined by a complexity measure, such as running time, of an algorithms that detemines whether a given string x belongs to language L.
Definition (Complexity class P)**.**

$$P = \{L \subseteq \{0, 1\}^* \mid \textit{ there exists an algorithm } A \textit{ that decides } L \textit{ in polynomial time}\}.$$

Lemma (P in terms of acceptance)**.**

$$\begin{aligned} P &= \{L \subseteq \{0,1\}^* \mid \textit{ there exists an algorithm } A \textit{ that decides } L \textit{ in polynomial time}\} \\ &= \{L \subseteq \{0,1\}^* \mid \textit{ there exists an algorithm } A \textit{ that accepts } L \textit{ in polynomial time}\}. \end{aligned}$$

We might not have an efficient algorithm that accepts a language L. Consider an algorithm A taking two parameters, $x, c \in \Sigma^*$. Instead of trying to find a solution to x (which may take a long time), A instead **verifies** that c is a solution to x.

Example 3 (The `HAM-CYCLE` problem)**.** *An undirected graph G is hamiltonian if it contains a simple cycle containing every vertex of G. We define*

$$\textit{HAM-CYCLE} = \{\langle G \rangle \mid G \textit{ is Hamiltonian}\}.$$

It is open whether `HAM-CYCLE` can be decided in polynomial time. However, it is easy to show that `HAM-CYCLE` can be verified in polynomial time:

Consider instead an algorithm A_{ham} taking two parameters, $\langle G \rangle$ and $\langle C \rangle$. A_{ham} checks that $\langle G \rangle$ defines an undirected graph G and that $\langle C \rangle$ encodes a cycle C containing every vertex of G exactly once. If so, $A_{ham} = 1$, and otherwise $A_{ham} = 0$, Designing A_{ham} to run in polynomial time is easy. Hence we can verify `HAM-CYCLE` in polynomial time.

Definition (Verification of language)**.** *A **verification algorithm** A is an algorithm taking two arguments $x, y \in \{0,1\}^*$, where y is the certificate and x is a string. Algorithm A **verifies** x if there is a certificate y such that $A(x, y) = 1$. The language verified by A is*

$$L = \{x \in \{0,1\}^* \mid \textit{ there is a } y \in \{0,1\}^* \textit{ such that } A(x, y) = 1\}.$$

For instance, from **Example 3** we have

$$\texttt{HAM-CYCLE} = \{x \in \{0,1\}^* \mid \text{ there is a } y \in \{0,1\}^* \text{ such that } A_{ham}(x, y) = 1\}.$$

Definition (Complexity class NP)**.** *NP is the class of languages that can be verified in polynomial time. More precisely, $L \in NP$ if and only if there is a polynomial-time verification algorithm A and a constant c such that*

$$L = \{x \in \{0,1\}^* \mid \textit{ there is a } y \in \{0,1\}^* \textit{ with } |y| = O(|x|^c) \textit{ such that } A(x, y) = 1\}.$$

We have that $P \subseteq NP$, but do not know if $P = NP$.

Definition (Complexity class co-NP)**.** *co-NP is the class of languages L such that $\bar{L} \in NP$.*

We do not know if $NP =$ co-NP. In words, for the `HAM-CYCLE` problem, given a graph, can we easily verify that it does *not* have a simple cycle containing every vertex of G? What is known is that $P \subseteq NP \cap$ co-NP.

There are problems in NP that are "the most difficult" in that class. If any one of them can be solved in polynomial time, then *every* problem in NP can be solved in polynomial time. These difficult problems are called NP-complete.

Definition (Reducibility)**.** *Language L_1 is polynomial-time **reducible** to language L_2 if there is a polynomial-time computible function $f : \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,*

$$x \in L_1 \iff f(x) \in L_2.$$

In this case, we write $L_1 \leq_P L_2$.

If $L_1 \leq_P L_2$ then L_1 is in a sense no harder to solve than L_2. More precisely,

$$L_1 \leq_P L_2 \wedge L_2 \in P \implies L_1 \in P.$$

This follows since any instance I_1 of L_1 can be solved by transforming it in polynomial time to an instance I_2 if L_2 and then solving I_2 with a polynomial-time algorithm for L_2.
Definition (NP-complete languages)**.** *A language L is NP-complete if*

1. $L \in NP$ *and*
2. $L' \leq_P L$ *for every* $L' \in NP$.

L is NP-hard if property 2 holds (and possibly not property 1). The class of NP-complete languages is denotes NPC. If some language of NPC belongs to P then $P = NP$ This is because any language in NP can then be transformed to this language in polynomial time, and then solved in polynomial time (making all NP languages solvable in polynomial time).

Technique for showing NP-completeness for a language L: Suppose L' is an NP-complete language. If $L' \leq_P L$ then L is NP-hard, because all other languages in NP can be reduced to L', which can in turn be reduced to L, i.e. all languages in NP can in this way be reduced to L. If it is also the case that $L \in NP$, then L is NP-complete (by the definition of NP-complete languages).

The method for showing $L \in NPC$ is thus as follows:

1. Prove $L \in NP$.
2. Select a known NP-complete language L'.
3. Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.
4. Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.
5. Prove that the algorithm computing f runs in polynomial time.

Problems in NP: `CIRCUIT-SAT`, `SAT`, `3-CNF-SAT`, `SUBSET-SUM`, `CLIQUE`, `VERTEX-COVER`, `HAM-CYCLE`, `TRAVELLING-SALESMAN-PROBLEM`.

6.2.1 Showing that VERTEX-COVER is NP-complete

Before we show that VERTEX-COVER is NP-complete, we first define the problem. A **vertex cover** of an undirected graph $G = (v, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then either $u \in V'$ or $u \in V'$ or both. That is, each $v \in V'$ covers all its incident edges, and a vertex cover for G covers all the edges E. The size of a vertex cover is the number of vertices in it. The **vertex-cover problem** is to find a vertex cover of minimum size in a given graph. This is an optimization problem. The corresponding decision problem is to determine whether a graph has a vertex cover of size k. We define VERTEX-COVER as the following language:

$$\{\langle G, k\rangle : \text{ graph } G \text{ has a vertex cover of size } k\}.$$

1. **Prove** $L \in NP$.

   We first prove that VERTEX-COVER is in NP, meaning that we can verify a potential solution in polynomial time. A certificate for vertex cover is the set of vertices $V' \subseteq V$; one can easily check whether V' is a vertex cover of size k in polynomial time, by first checking that $|V'| = k$ and then going through all edges in E and checking that each edge is incident to at least one vertex in V'.

2. **Select a known NP-complete language L'.**

   We select CLIQUE.

3. **Describe an algorithm that computes a function f mapping every instance $x \in \{0,1\}^*$ of L' to an instance $f(x)$ of L.**

   The reduction from CLIQUE to VERTEX-COVER relies on the notion of the "complement" of a graph. Given an undirected graph $G = (V, E)$, the **complement** of G is $\bar{G} = (V, \bar{E})$, where $\bar{E} = \{(u, v) : u, v \in V, u \neq v, \text{ and } (u, v) \notin E\}$. That is $\bar{G}$ is the graph containing exactly the edges not in G.

   The reduction takes an instance $\langle G, k\rangle$ of CLIQUE as input. It then computes $\bar{G}$ in polynomial time and returns the instance $\langle \bar{G}, |V| - k\rangle$ of the vertex-cover problem.

4. **Prove that the function f satisfies $x \in L'$ if and only if $f(x) \in L$ for all $x \in \{0,1\}^*$.**

   We now have to prove that: The graph G has a clique of size k if and only if the graph $\bar{G}$ has a vertex cover of size $|V| - k$. To show that the first implies the second, suppose that G has a clique $V' \subseteq V$ with $|V'| = k$. Then $V - V'$ is a vertex cover in $\bar{G}$. Let (u, v) be any edge in $\bar{E}$. Then $(u, v) \notin E$, which implies that at least one of u or v does not belong to V', since every pair of vertices in V' is connected by an edge of E. This means that at least one of u or v is in $V - V'$, which means that edge (u, v) is covered by $V - V'$. Since (u, v) was chosen arbitrarily from $\bar{E}$, every edge of $\bar{E}$ is covered by a vertex in $V - V'$. Hence, the set $V - V'$ which has size $|V| - k$, forms a vertex cover for $\bar{G}$.

   Conversely, suppose that $\bar{G}$ has a vertex cover $V' \subseteq V$, where $|V'| = |V| - k$. Then, for all $u, v \in V$, if $(u, v) \in \bar{E}$, then $u \in V'$ or $v \in V'$ or both. The contrapositive of this implication is that for all $u, v \in V$, if $u \notin V'$ and $v \notin V'$, then $(u, v) \in E$. In other words $V - V'$ is a clique, and it has size $|V| - |V'| = k$.

5. **Prove that the algorithm computing f runs in polynomial time**.

   We can easily compute the complement of G in polynomial time by going through $|V|^2$ iterations, choosing any two distinct vertices u and v, and adding an edge $\{u, v\}$ to $\bar{E}$ if $\{u, v\} \notin E$.

6.2.2 Showing that TSP is NP-complete

Problem definition: A Hamiltonian cycle is a path in an undirected graph where each vertex is visited exactly once. In the **travelling salesman problem**, a salesman must visit n cities. Modeling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a **tour**, or Hamiltonian cycle, visiting each city exactly once, and finishing in the city he started in. The salesman incurs a nonnegative integer cost $c(i, j)$ to travel from city i to city j, and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour. The formal language for the corresponding decision problem is

$$\begin{aligned}\texttt{TSP} = \{\langle G, c, k\rangle : {} & G = (V, E) \text{ is a complete graph,} \\ & c \text{ is a function from } V \times V \longrightarrow \mathbb{N}, \\ & k \in \mathbb{N}, \text{ and,} \\ & G \text{ has a traveling-salesman tour with cost at most } k.\end{aligned}$$

We now want to prove that TSP is NP-complete by reducing a known NP-complete problem, namely the HAM-CYCLE problem, to TSP in polynomial time.

1. **Prove** $L \in NP$.

   A certificate is a sequence of the n vertices in the tour. The verification checks that each vertex is contained once in the sequence, sums the edge costs and checks that the sum is at most k. This is easily done in polynomial time.

2. **Select a known** NP**-complete language** L'.

   As mentioned, we select the HAM-CYCLE problem, which is NP-complete.

3. **Describe an algorithm that computes a function** f **mapping every instance** $x \in \{0, 1\}^*$ **of** L' **to an instance** $f(x)$ **of** L.

   Let $G = (V, E)$ be an instance of HAM-CYCLE. We construct an instance of TSP as follows: We form the complete graph $G' = (V, E')$, where $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$, and we define the cost function c by

   $$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E, \\ 1 & \text{if } (i, j) \notin E. \end{cases}$$

   The instance of TSP is then $\langle G', c, 0\rangle$.

4. **Prove that the function** f **satisfies** $x \in L'$ **if and only if** $f(x) \in L$ **for all** $x \in \{0, 1\}^*$.

   Suppose that graph G has a Hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since the costs of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E. We conclude that h' is a Hamiltonian cycle in graph G.

5. **Prove that the algorithm computing** f **runs in polynomial time**.

   Clearly, the above can be done in polynomial time.

6.3 Short Notes

An **alphabet** is a finite set Σ of symbols. A **language** L over alphabet Σ is a set of strings of symbols from Σ. We use $\Sigma = \{0, 1\}$, so L is a set of binary strings. We denote Σ^* as the set of all binary strings, including the empty string ϵ.

A **decision problem** Q consists of yes-instances and no-instances. We can view Q as a mapping of yes-instances to 1 and no-instances to 0. We can also view Q as a language L:

$$L = \{x \in \{0, 1\}^* \mid Q(x) = 1\}.$$

A **verification algorithm** as an algorithm A taking two arguments, $x, y \in \{0, 1\}^*$, where y is the certificate. A verifies string x if there is a certificate y such that $A(x, y) = 1$. The language L verified by A is

$$L = \{x \in \{0, 1\}^* \mid \text{ there is a } y \in \{0, 1\}^* \text{ such that } A(x, y) = 1\}.$$

NP is the class of languages that can be verified in polynomial time. In other words, $L \in NP$ if and only if there is a polynomial-time verification algorithm A and a constant c such that

$$L = \{x \in \{0, 1\}^* \mid \text{ there is a } y \in \{0, 1\}^* \text{ with } |y| = O(|x|^c) \text{ such that } A(x, y) = 1\}.$$

Language L_1 is polynomial-time **reducible** to language L_2 if there is a polynomial-time computible function $f : \{0, 1\}^* \longrightarrow \{0, 1\}^*$ such that for all $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2.$$

This is denoted $L_1 \leq_P L_2$. it follows that if L_1 can be reduced to L_2 in polynomial-time and L_2 is deciable in polynomial-time, then L_1 is also deciable in polynomial-time:

$$L_1 \leq_P L_2 \wedge L_2 \in P \implies L_1 \in P.$$

Language L is NP-**complete** if

1. $L \in NP$ and
2. $L' \leq_P L$ for every $L' \in NP$.

We say that L is NP-**hard** if it satisfies property 2. If any language of NPC belongs to P, then $P = NP$.

7 Exact Exponential Algorithms and Parameterized Complexity

7.1 Disposition

(1) Sketch presentation outline.

(2) Motivation: Why are exact exponential algorithms and parameterized complexity interesting.

(3) Exact exponential algorithm example: Travelling-Salesman Problem.

(4) Parameterized complexity example: Vertex-Cover Problem.

7.2 Comprehensive Notes

Definition (O^* notation)**.** *For functions f and g we write $f(n) = O^*(g(n))$ iff $f(n) = O(g(n)poly(n))$, where $poly(n)$ is a polynomial.*

For example, for $f(n) = 2^n n^2$ and $g(n) = 2^n$, $f(n) = O^*(g(n))$.

Measuring quality of exact algorithms: The running time of an algorithm is estimated by a function either of the input length or of the input "size". The input length can be defined as the number of bits in any reasonable encoding of the input over a finite alphabet; the notion of input size is problem dependent.

7.2.1 Exact Exponential Algorithms (EEA)

7.2.2 Parameterized Complexity (PC)

Parameterized complexity measures complexity not only in terms of input length but also in terms of a parameter which is a numerical value not necessarily dependent on the input length. Many parameterized algorithmic techniques evolved accompanied by a powerful complexity theory. It seeks the possibility of obtaining algorithms whose running time can be bounded by a polynomial function of the input length and, usually, an exponential function of the parameter. Thus most of the exact exponential algorithms studied in this book can be treated as parameterized algorithms, where the parameter can be the number of vertices in a graph, etc.

Similarities between EEA and PC: Many basic techniques, such as branching dynamic programming, iterative compression and inclusion-exclusion are used in both areas. There are also very nice connections between sub-exponential complexity and parameterized complexity.
Definition (Fixed-Parameter Algorithms, or FPT)**.** *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, trying to make both the $f(k)$ factor and the constant c in the bound on the running time as small as possible. FPT algorithms can be put in contrast with less efficient XP algorithms, where the running time is of the form $f(k) \cdot n^{g(k)}$, for some function f and g. There is a tremendous difference in the running times.*

*in parameterized algorithmics, k is simply a **relevant secondary measurement** that encapsulates some aspect of the input instance, be it the size of the solution sought after, or a number describing how "structured" the input instance is.*
Definition (A 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**.*

7.2.3 Dynamic Programming for TSP

Definition (Travelling Salesman Problem)**.** *Given a set of distinct cities* $\{c_1, c_2, \dots, c_n\}$ *and for each pair* $c_i \neq c_j$*, the distance between* c_i *and* c_j *is denoted by* $d(c_i, c_j)$*. The task is to construct a tour of the travelling salesman of minimum total length which visits all the cities and returns to the starting point. In other words, the task is to fund a permutation* π *of* $\{1, 2, \dots, n\}$*, such that the following sum is minimized:*

$$\left(\sum_{i=1}^{n-1} d\left(c_{\pi(i)}, c_{\pi(i+1)}\right)\right) + d\left(c_{\pi(n)}, c_{\pi(1)}\right)$$

The naive brute force approach is to enumerate all possible permutations of $1, 2, \dots, n$, of which there are $n!$. Using dynamic programming one obtains a much faster algorithm.

The idea is as follows: For every pair (S, c_i), where S is a nonempty subset of $\{c_2, c_3, \dots, c_n\}$ and $c_i \in S$, the algorithm computes the value $OPT[S, c_i]$, which is the minimum length of a tour which starts in c_1, visits all cities from S and ends in c_i.

We compute the values $OPT[S, c_i]$ dynamically and bottom-up, in order of increasing cardinality of S. The computation of $OPT[S, c_i]$ in the case S contains only one city is trivial, because in this case, $OPT[S, c_i] = d(c_1, c_i)$. For the case $|S| > 1$, the value $OPT[S, c_i]$ can be expressed in terms of subsets of S:

$$OPT[S, c_i] = \min\left\{ OPT\left[S \setminus \{c_i\}, c_j\right] + d\left(c_j, c_i\right) \,:\, c_j \in S \setminus \{c_i\} \right\}. \tag{2}$$

Indeed, if in some optimal tour in S terminating in c_i, the city c_j immediately precedes c_i, then

$$OPT[S, c_i] = OPT\left[S \setminus \{c_i\}, c_j\right] + d\left(c_j, c_i\right).$$

Thus taking the minimum over all cities that can precede c_i, we obtain (2). Finally, the value OPT of the optimal solution is the minimum of

$$OPT\left[\{c_2, c_3, \dots, c_n\}, c_i\right] + d\left(c_i, c_1\right),$$

where the minimum is taken over all indices $i \in \{2, 3, \dots, n\}$.

The amount of steps required to compute (2), for a fixed set S of size k and all vertices $c_i \in S$ is $O(k^2)$, since for a specific c_i, we iterate over all $c_j \in S \setminus \{c_i\}$ to find the optimal preceding node. We compute (2) for every subset S of cities, and thus takes time $\sum_{k=1}^{n-1} O\left(\binom{n}{k}\right)$. Therefore, the total time to compute OPT is

$$\sum_{k=1}^{n-1} O\left(\binom{k}{n} k^2\right) = O\left(n^2 2^n\right).$$

The improvement from $O(n!n)$ in the trivial enumeration algorithm to $O^*(2^n)$ in the dynamic programming algorithm is quite significant.

However, the algorithm is exponential in space as well as time, meaning that a $O(2^n)$ memory is used as well.

7.2.4 Bar-Fight Prevention a.k.a Vertex Cover

The problem: You are a bouncer that wants to preemptively stop fights at a bar. Denote a graph G of n nodes, where nodes represent bar guests and edges are fights, i.e. guest a will fight guest b if they share an edge. You can at most reject k guests from the bar. Thus, this problems translates to: Is there a vertex cover of G of size at most k?

Unfortunately, this is NP-complete, and so the naive approach, trying all possibilities, runs in $O(2^n)$ time. However,by restricting the parameter k, better solutions may be found.

Insight: You can always add a node with degree $d \geq k + 1$ to your vertex cover, since if you did not, you would have to add all $k + 1$ neighbours to your vertex cover, breaking the bound in the cover size.

After doing this, all remaining nodes will have at most degree k.

Insight: Every edge has to be covered, and the only to do this is to include one of its endpoints in the vertex cover. Thus, we can proceed by doing the following: For a given edge $\{u, v\}$, try adding u to the vertex cover and run the algorithm recursively to check whether the remaining edges can be covered using at most $k - 1$ vertices. If this succeeds, we have a solution. If not, then replace u by v in the vertex cover and run the algorithm recursively to check whether the remaining edges can be covered using at most $k - 1$ vertices. If this also fails, then you are guaranteed that no vertex cover of size k exists.

What is the running time? For each recursive call, we decrease k by 1. When k reaches 0, all the algorithm has to do is to check if there are remaining edges in the graph not covered by the proposed vertex cover. Each call spawn two recursive calls, which we do at most k times, giving a total of at most 2^k recursive calls. Let m be the number of edges. Each call runs in linear time $O(n + m)$, since we have to check if all edges are covered.

We know something about the number of edges m. Since each node has at most k edges (because we removed nodes with more edges), there is at most $\frac{nk}{2}$ edges in the graph. Thus $m = O(nk)$ and $O(n + m) = O(nk)$.

The total runtime is thus $O(2^k nk)$. The naive algorithm that tries all possible subsets of k people runs in time $O(n^k)$.

7.3 Presentation

To begin with, under the assumption that $P \neq NP$, there are no polynomial-time, exact algorithms for any NP-hard problem. Although we don't know whether $P = N$ or $P \neq NP$, we don't have any polynomial-time algorithms for any NP-hard problems.

The idea behind fixed-parameter tractability is to take an NP-hard problem, which we don't know any polynomial-time algorithms for, and to try to separate out the complexity into two pieces - some piece that depends purely on the size of the input, and some piece that depends on some "parameter" to the problem.

8 Approximation Algorithms

8.1 Disposition

(1) Sketch presentation outline.

(2) Motivation: Why are approximation algorithms interesting.

(3) Definition of approximation ratio:

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

(4) Example: Vertex cover.

(5) Example: 3-CNF-SAT.

8.2 Comprehensive Notes

Many problems of practical significance are NP-complete, yet they are too important to abandon merely because we do not know how to find an optimal solution in polynomial time. We have at least three ways to get around NP-completeness: (1) For small cases, exponential running time might be perfectly OK, (2) we might be able to restrict our problem to special cases which we can solve faster, or (3) we might be OK with a solution that is near-optimal. We call an algorithm that returns near-optimal solutions an **approximation algorithm**.

Definition (Approximation ratio, $\rho(n)$-approximation)**.** *We say that an algorithm for a problem has an* ***approximation ratio*** *of $\rho(n)$ if, for any input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:*

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n).$$

*If a algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-**approximation algorithm**. The definitions of the approximation ratio and of a $\rho(n)$-approximation algorithm apply to both minimization and maximization problems.*

Definition (Approximation scheme)**.** *An* ***approximation scheme*** *for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\epsilon > 0$ such that for any fixed ϵ, the scheme is a $(1+\epsilon)$-approximation algorithm.*

We say that an approximation scheme is a ***polynomial-time approximation scheme*** *if for any fixed $\epsilon > 0$, the scheme runs in time polynomial in the size n of its input.*

We say that an approximation scheme is a ***fully polynomial-time approximation scheme*** *if it is an approximation scheme and its running time is polynomial in both $\frac{1}{\epsilon}$ and the size n of the input instance.*

E.g. a scheme with running time $O((1/\epsilon)^2 n^3)$ is such a scheme, while $O(n^{2/\epsilon})$ is not.

8.2.1 The Vertex-Cover Problem

The following algorithm is a polynomial-time 2-approximation algorithm:

Algorithm 3: APPROX-VERTEX-COVER

Data: An undirected graph G with n vertices and edges E.
Result: A vertex cover with size no more than twice the size of the optimal vertex cover.

1 $C = \emptyset$
2 $E' = E$
3 **while** $E' \neq \emptyset$ **do**
4 let (u, v) be an arbitrary edge of E'
5 $C = C \cup \{u, v\}$
6 remove from E' every edge incident on either u or v.
7 **return** C

Proof. The running time is $O(n + |E|)$, since we go through all edges of E and potentially adds n nodes to C. This is polynomial running-time.

The set C is a vertex cover, since the algorithm goes through all edges, only removing a given edge e after a vertex has been found (and added to C) which covers e.

Let A denote the set of edges that are chosen arbitrarily from E' (line 4). Any vertex cover, including the optimal cover C^*, must cover the edges in A, meaning that for each edge in A, either of its endpoints must be in the vertex cover. Since no two edges in A share endpoints, at least one node for each edge must be in a vertex cover, and we have a lower bound of

$$|A| \leq |C^*|$$

In addition, by construction we have that

$$|C| = 2|A|,$$

from which we can now bound the approximated solution by

$$|C| = 2|A| \leq 2|C^*|.$$

□

8.2.2 Randomized Approximation Algorithm for MAX-3-CNF Satisfiability

We say that a randomized algorithm for a problem has an **approximation ratio of** $\rho(n)$ if, for any input of size n, the *expected* cost C of the solution produced by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution. That is, we look at expected cost in case of randomized algorithms.

In **MAX-3-CNF satisfiability**, the input is the same as for 3-CNF satisfiability, and the goal is to return an assignment of the variables that maximize the number of clauses evaluating to 1.

We now show that randomly flipping a coin for each variable, giving equal change of it being 1 or 0, yields a randomized $8/7$-approximation algorithm.

Theorem. *Given an instance of MAX-3-CNF satisfiability with n variables $x_1, x_2, \ldots, x_n$ and m clauses, the randomized algorithm that independently sets each variable to 1 with probability $1/2$ and to 0 with probability $1/2$ is a randomized $8/7$-approximation algorithm.*

Proof. Suppose that we have independently flipped a coin for each variable. For $i = 1, 2, \ldots, m$, we define the indicator random variable

$$Y_i = \begin{cases} 1, & \text{if clause } i \text{ is satisfied} \\ 0 & \text{otherwise,} \end{cases}$$

so that $Y_i = 1$ as long as we have set at least one of the literals in the ith clause to 1. Since no literal appears more than once in each clause, and since we have assumed that no variable and its negation appear in the same clause, the settings of the three literals in each clause are independent. A clause is not satisfied only if all three of its literals are set to 0, and so $\Pr[\text{clause } i \text{ is not satisfied}] = \left(\frac{1}{2}\right)^3 = \frac{1}{8}$. Thus we have $\mathbb{E}[Y_i] = 1 - \frac{1}{8} = \frac{7}{8}$.

Let $Y = Y_1 + Y_2 + \ldots + Y_m$. Then we have

$$\begin{aligned} \mathbb{E}[Y] &= \mathbb{E}\left[\sum_{i=1}^{m} Y_i\right] \\ &= \sum_{i=1}^{m} \mathbb{E}[Y_i] && \text{by linearity of expectation} \\ &= \sum_{i=1}^{m} \frac{7}{8} \\ &= \frac{7m}{8}. \end{aligned}$$

Clearly, m is an upper bound on the number of satisfied clauses, since m is the number of clauses, and thus we have $C^* \leq m$. We have also just shown that $C = \frac{7m}{8}$ in expectation. Hence the approximation ratio is at most

$$\frac{C^*}{C} \leq \frac{m}{\frac{7m}{8}} = \frac{8}{7}.$$

□

8.3 Presentation

1. **Motivation:** As we know from NP-completeness, some problems are harder to solve than others. Unfortunately, there are problems which we cannot hope to solve in polynomial time that are still practically interesting, and thus we cannot just ignore them.

   One way to deal with these kinds of problems is to make do with near optimal solutions instead of exact optimal solutions, which is what approximation algorithms do. We make a trade off between the exactness of the solution, with the running time of the solution. Thus, approximation algorithms are faster, but inaccurate to some extent.

2. In order to give bound on how far away from the optimal solution an approximation is, we use the concept of $\rho(n)$-approximations. That is, for some input instance of size n, the approximated solution is at most some factor $\rho(n)$ worse than the optimal solution.

   We can only approximate solutions for optimization problems, and not decision problems.

   We measure this by
   $$\max\left\{\frac{C}{C^*}, \frac{C^*}{C}\right\} \leq \rho(n).$$
   where the largest fraction is determined by whether the problem is a maximization problem or a minimization problem.

3. OK, so now let me give some examples of approximation algorithms. The first I want to show is a 2-approximation of the travelling salesman problem. I have to show to things: One, that the algorithm is fast, meaning that it runs in polynomial time, and Two, that the approximated solution is at most twice as bad as the optimal solution.

References

[1] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2009.