Advanced Algorithms and Datastructures - Exam Notes

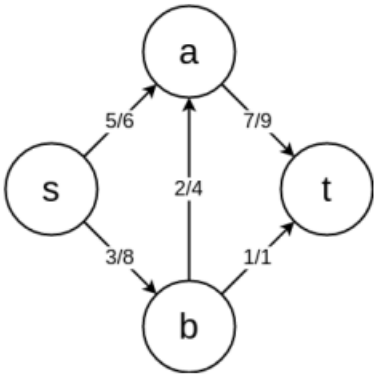
André O. Andersen

2021

Max Flow

Disposition

1. Introduction
2. Introduction to
    - *Flow network*
    - *Residual network*
    - *Ford-Fulkerson + Edmonds-Karp*
3. Example of running the Edmonds-Karp Algorithm
4. Proof of the *Max-flow min-cut theorem*



Presentation

Hey guys. I will be talking about Max flow. I have here a disposition of the things I will go through *Hand out disposition*. As you can see, mine is identical to yours and it does not contain any information that it is not allowed to *Show own disposition*. First I will be giving a brief introduction to the topic, then I will be running the Edmonds-Karp algorithm on the example at the bottom of the page. Lastly, I will be proving the so-called Max-Flow min-cut theorem.

In Max-flow we are given a ***flow network***, which is a directed graph where each edge $(u, v) \in E$ has a nonnegative ***capacity*** $c(u, v) \geq 0$. We require that for each edge, the antiparallel edge does not exist. We distinguish two vertices; a ***source*** s and a ***sink*** t.

A ***flow*** in G is a function $f : V \times V \to \mathbb{R}$ that satisfies the following two properties:

1. ***Capacity constraint***: For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$
2. ***Flow conservation***; For all $u \in v - \{s, t\}$ we require

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v).$$

The goal is to maximize the value $|f|$ defined by

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

without violating any of the constraints.

To solve this problem we use a ***residual network***, which keeps track of where it is possible to add flow. This is induced by applying

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{if } (u, v) \in E \\ f(v, u) & \text{if } (v, u) \in E \\ 0 & \text{otherwise.} \end{cases}$$

to each of the edges in G.

To solve this problem one can make use of the Ford-Fulkeson method. It iteratively increases the value of the flow. We start with $f(u, v) = 0$ for all $u, v \in V$. At each iteration, we increase the flow value in G by finding an augmenting path in an associated ***residual network*** G_f. Once we know the edges of an augmenting path p in G_f, we can define the ***residual capacity*** of the path p by

$$c_f(p) = \min\{c_f(u, v) : (u, v) \text{ is on } p\}$$

and the flow $f_p : V \times V \to \mathbb{R}$ in G_f by

$$f_p(u, v) = \begin{cases} c_f(p) & \text{if } (u, v) \in p \\ 0 & \text{otherwise.} \end{cases}$$

We can then use this to augment the flow f by f_p to get closer to maximum, by

$$(f \uparrow f_p)(u, v) = \begin{cases} f(u, v) + f_p(u, v) - f_p(v, u) & \text{if } (u, v) \in E \\ 0 & \text{otherwise.} \end{cases}$$

We repeatedly change the flow until the residual network has no more augmenting paths. The max-flow min-cut theorem tells us, that upon termination, this process yields a maximum flow.

One implmentation of Ford-Fulkerson is the ***Edmonds-Karp Algorithm***, which in each iteration finds the augmenting path by using breadth-first search.

I will now run the algorithm on the example *Run Edmonds-Karp*... and since there are no augmenting paths, max-flow min-cut theorem tells us, that f is a maximum flow in G.

Now that I have shown how the Edmonds-Karp algorithm works, I will now prove the Max-Flow min-cut theorem. First, let's recap what the theorem tells us. The theorem says, that if f is a flow in a flow network $G = (V, E)$ with source s and a sink t, then the following cases are equivalent.

1. f is a maximum flow in G
2. The residual network G_f contains no augmenting paths
3. $|f| = c(s, T)$ for some cut (S, T) of G.

We can now prove the max-flow min-cut theorem. The proof works by proving that the cases imply each other. We start of by proving that the first case implies the second case:

> Suppose that f is a maximum flow in G but that G_f has an augmenting path p. Then the flow found by augmenting f by f_p is a flow in G with value strictly greater than $|f|$, contradicting the assumption that f is a maximum flow.

Now, let's prove that the second case implies the third case:

> Suppose that G_f has no augmenting path. Define
>
> $$S = \{v \in V : \text{ there exists a path from } s \text{ to } v \text{ in } G_f\}$$
>
> and $T = V - S$. The partition (S, T) is then a cut. Now, consider a pair of vertices $u \in S$ and $v \in T$. If $(u, v) \in E$, we must have $f(u, v) = c(u, v)$. If $(v, u) \in E$, we must have $f(v, u) = 0$. If neither (u, v) nor (v, u) is in E, then $f(u, v) = f(v, u) = 0$. We thus have
>
> $$\begin{aligned} f(S,T) &= \sum_{u \in S} \sum_{v \in t} f(u,v) - \sum_{u \in S} \sum_{v \in t} f(v,u) \\ &= \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{u \in S} \sum_{v \in T} 0 \\ &= c(S,T). \end{aligned}$$
>
> From a lemma we have that $|f| = f(S, T)$. Thus, we have $|f| = f(S, T) = c(S, T)$.

Lastly, let's prove that the third case implies the first case:

> We have that $|f| \leq c(S, T)$ for all cuts (S, T). The condition $|f| = c(S, T)$ thus implies that f is a maximum flow.

Extra

A cut (S, T) is a partition of V into S and $T = V - S$, such that $s \in S$ and $t \in T$. The capacity of the cut $c(S, T)$ is then just simply the sum of the capacities of the edges going across the cut from S to T

$$c(S,T) = \sum_{u \in S} \sum_{v \in T} c(u,v),$$

and the ***net flow*** $f(S, T)$ across the cut is sum of the flows of the edges going across the cut from S to T minus the sum of the flows of the edges going across the cut from T to S

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u).$$

We also have, that $f(S,T) = |f|$.

Let $G = (V,E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Let $G = (V,E)$ be a flow network, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in equation (**??**), and suppose that we augment f by f_p. then the function $f \uparrow f_p$ is a flow in G with value $|f \uparrow f_p| = |f| + |f_p| > |f|$.

Questions they can ask

Why do we require that antiparallel edges do not exist?

I think it is because, that we in the residual network have antiparallel edges. If we then also had antiparallel edges in the flow network, something would probably screw up.

How do we handle antiparallel edges?

We split one of the edges up into two edges and introduce a new vertex, which connects the two new edges. The capacity of the two new edges is the same as the capacity of the original edge.

Why do we have flow going in the wrong direction in the residual network?

Sending flow back along an edge is equivalent to decreasing the flow on the edge, which is a necessary operation in many algorithms

Does the algorithm always terminate?

No. The Ford-Fulkerson method might fail if edge capacities are irrational numbers.

What is the running time of the algorithm?

This depends on how we find the augmenting path. If we assume that the capacities are integers, then Ford-Fulkerson has a running time of $O(E \cdot |f^*|)$, where f^* is a max flow. This is because the capacities are integers, we know Ford-Fulkerson increases the value by atleast 1 at each step, making the runtime be bounded by the max flow ($|f^*|$) and E is used for finding an augmenting path.

We can improve the bound on Ford-Fulkerson by finding the augmenting path with a breadth-first search. This has arunning time of $O(VE^2)$.

Linear Programming and Optimization

Program

1. Introduction
2. Introduction to
   - Various forms of LP
   - *Objective function*
   - *Constraints*
   - Algorithm for solving linear programming
3. Preparing and running SIMPLEX on example
4. Duality:
   - Motivation behind duality
   - Definition of *dual linear program*
   - Definition of *weak duality*
   - Proof of weak duality

$$\begin{array}{ll} \text{minimize} & -x_1 - 2x_2 \\ \text{subject to} & x_1 + x_2 = 6 \\ & x_1 - 2x_2 \leq 4 \\ & x_1 \geq 0 \end{array}$$

Presentation

Hey guys. I will be talking about Linear Programming. I have here a program of the things I will go through *Hand out program*. As you can see, mine is identical to yours and it does not contain any information that it is not allowed to *Show own program*. First I will be giving a brief introduction to the topic, then I will transform an example into standard form, which will then be transformed into slack form, which I then will run the SIMPLEX-algorithm on. Lastly, I will pove *weak duality*, which says, that the solution to the dual of a linear program is always an upper bound on the solution to the orignial linear program.

Linear programming is given in two forms; ***standard form*** and ***slack form***.

standard form has the following form

$$\sum_{j=1}^{n} c_j x_j$$

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad \text{for } i = 1, 2, ..., m$$

$$x_j \geq 0 \quad \text{for } j = 1, 2, ..., n$$

The first equation is the ***objective function***, which we wish to optimize with respect to the ***constraints***, which are the following lines, where the last constraint is required in standard form.

Slack form is very similar to standard form, however, only the nonnegativity constraints are the only inequality constraints, and the remaining constraints are equalities.

To solve a linear program we use the simplex algorithm, whose input is a linear program in ***slack form***.

Let's transform the example into standard and slack form *Write the example on the blackboard*. First off, we see that the example is a minimization problem instead of a maximization problem. This can easily be fixed by negating the coefficients in the objective function. Thus, by doing so we obtain the new objective function

$$x_1 + x_2.$$

Since the two linear programs have identical sets of feasible solutions and, for any feasible solution, the objective value the first linear program is the negative of the objective value in the second program, the two linear programs are equivalent.

Next, we see that x_2 does not have a nonnegativity constraint. We can fix this by replacing each occurence of x_2 by $x_2' - x_2''$ and add the nonnegativity constraint $x_2', x_2'' \geq 0$. To ease the readibility, I will instead let $x_2' = x_2$ and $x_2'' = x_3$. Thus, by doing so we receive the objective function $x_1 + 2x_2 - 2x_3$ the two constraints

$$x_1 + x_2 - x_3 = 6$$

and

$$x_1 - 2x_2 + 2x_3 \leq 4$$

and the nonnegativity constraint

$$x_1, x_2, x_3 \geq 0.$$

Any feasible solution $\hat{x}$ to the new linear program corresponds to a feasible solution $\bar{x}$ to the original linear program with $\hat{x}_j = \hat{x}_j' - \hat{x}_j''$ and with the same objective value. Also, any feasible solution $\hat{x}$ to the original linear program corresponds to a feasible solution $\hat{x}$ to the new linear program with $\hat{x}_j' = \bar{x}_j$ and $\hat{x}_j'' = 0$ if $\hat{x}_j \geq 0$, or with $\hat{x}_j'' = -\bar{x}_j$ and $\hat{x}_j' = 0$ if $\bar{x}_j < 0$. Thus, the two linear programs are equivalent.

We also see, that the first constraint has an equal sign instead of an $\geq$. We know that the equality only

holds if and only if both $\geq$ holds and $\leq$ holds. Thus, we can replace the equality constraint by the pair of inequality constraints that uses $\leq$ and $\geq$ instead. Thus, we replace the constraint

$$x_1 + x_2 - x_3 = 6$$

with the two constraints

$$x_1 + x_2 - x_3 \leq 6$$

and

$$x_1 + x_2 - x_3 \geq 6$$

Lastly, we see, that the constraint

$$x_1 + x_2 - x_3 \geq 6$$

has a greater-than-or-equal-to-sign instead of a less-than-or-equal-to-sign. This can easily be fixed by multiplying the constraint through by -1. Thus, we instead obtain

$$-x_1 - x_2 + x_3 \geq -6$$

In total we have

$$\begin{array}{ll} \text{maximize} & x_1 + 2x_2 - 2x_3 \\ \text{subject to} & x_1 + x_2 - x_3 \leq 6 \\ & -x_1 - x_2 + x_3 \geq -6 \\ & x_1 - 2x_2 + x_3 \leq 4 \\ & x_1, x_2, x_3 \geq 0 \end{array}$$

The linear program is now in standard form. Let transform this further into slack form. This is rather easily done by introducing a slack variable for each constraint, which measures the difference in the left and right hand side. This is done by subtracting the left hand side on both sides and introducing a new nonnegativity constraint for each constraint. By doing so we get the linear programming

$$\begin{aligned} z &= x_1 + 2x_2 - 2x_3 \\ x_4 &= 6 - x_1 - x_2 + x_3 \\ x_5 &= -6 + x_1 + x_2 - x_3 \\ x_6 &= 4 - x_1 + 2x_2 - x_3 \end{aligned}$$

where we have omitted the nonnegativity constraints.
Thus, the linear program is now in slack form and we can perform the SIMPLEX algorithm. The simplex algorithm works by continuously changing the solution by pivoting variables to and from the basic variables, which are the variables on the left hand side. This is done by picking a variable in the objective function which positively increases the value, and pivoting it with the basic variable that bottlenecks how much the non-basic variable can be increased. Let's run the SIMPLEX algorithm. We have

$$\begin{aligned} z &= x_1 + 2x_2 - 2x_3 \\ x_4 &= 6 - x_1 - x_2 + x_3 \\ x_5 &= -6 + x_1 + x_2 - x_3 \\ x_6 &= 4 - x_1 + 2x_2 - x_3 \end{aligned}$$

We see, that x_1 and x_2 are the only variables that can increase the value of the objective function. We choose x_2 as the pivot element (**DETTE ER VIGTIGT FOR AT FÅ ALGORTIMEN TIL AT TERMINERE EFTER 1 ITERATION - ALT ANDET VIL FÅ DEN TIL AT KØRE LÆNGERE**). Next, we isolate x_2 in x_4, since x_4 is the only constraint with a negative coefficient for x_2. The value of x_2 is then inserted in the other constraints as well as the objective function, resulting in

$$\begin{aligned} z &= 12 - x_1 - 2x_4 \\ x_2 &= 6 - x_1 + x_3 - x_4 \\ x_5 &= -x_4 \\ x_6 &= 16 - 3x_1. \end{aligned}$$

We now stop, as no basic variable appear in the objective function with a positive coefficient. Our solution is this $(0, 6, 0, 0, 0, 16)$, or $(x_1, x_2) = (0, 6)$ for the original problem, with a value of 12. Now that we have run the algorithm, I will be proving *weak duality*. Weak duality is a weak version of something called *duality*, which is used to prove that a solution is optimal. Given a linear program in standard form we define the dual linear program as

$$\begin{aligned}
&\text{minimize} \\
&\qquad \sum_{i=1}^{m} b_i y_i \\
&\text{subject to} \\
&\qquad \sum_{i=1}^{m} a_{ij} y_i \geq c_j && \text{for } j = 1, 2, ..., n \\
&\qquad y_i \geq 0 && \text{for } i = 1, 2, ..., m.
\end{aligned}$$

In weak duality we let $\bar{x}$ be a feasible solution to a primal, that is the "original", linear program in standard form and let $\bar{y}$ be any feasible solution to the corresponding linear program. Then, we have

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

because

$$\begin{aligned}
\sum_{j=1}^{n} c_j \bar{x}_j &\leq \sum_{j=1}^{m} \left(\sum_{i=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j && (\text{Using } \sum_{i=1}^{m} a_{ij} y_i \geq c_j) \\
&= \sum_{i=1}^{m} \left(\sum_{j=1}^{n} a_{ij} \bar{x}_j \right) \bar{y}_i \\
&\leq \sum_{i=1}^{m} b_i \bar{y}_i && (\text{Using } \sum_{j=1}^{n} a_{ij} x_j \leq b_i)
\end{aligned}$$

Extras

We say that two maximization linear programs L and L' are ***equivalent*** if for each feasible solution $\bar{x}$ to L with objective value z, there is a corresponding feasible solution $\bar{x}'$ to L' with objective value z, and for each feasible solution $\bar{x}'$ to L' with objective value z, there is a corresponding feasible solution $\bar{x}$ to L with objective value z. A minimization linear program L and a maximization linear program L' are equivalent if for each feasible soltuion $\bar{x}$ to L with objective value z, there is a corresponding feasible solution $\bar{x}'$ to L' with objective value $-z$, and for each feasible solution $\bar{x}'$ to L' with objective value z, there is a corresponding feasible solution $\bar{x}$ to L with objective value $-z$.

Corollary 29.9

Let $\bar{x}$be a feasible solution to aprimal linear program, and let $\bar{y}$ be a feasible solution to the corresponding dual linear program. If

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

then $\bar{x}$ and $\bar{y}$ are optimal solutions to the primal and dual linear programs, respectively.
Proof By the lemma of weak duality, the objective value of a feasible solution to the primal cannot exceed that of a feasible solution to the dual. The primal linear program is a maximization problem and the dual is a minimization problem. Thus, if feasible solutions $\bar{x}$ and $\bar{y}$ have the same objective value, neither can be improved.

Questions they can ask

ER DER NOGET HER?

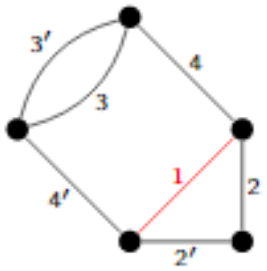
Randomized Algorithms

Disposition

1. Introduction
2. Quicksort:
   (a) Idea behind Quicksort
   (b) Example on running non-random Quicksort
   (c) Motivation behind randomness in Quicksort
   (d) Analysis of expected runtime of randomized quicksort
3. Min-cut:
   (a) Introduction to min-cut
   (b) Example on running the min-cut algorithm
4. Las Vegas Algorithms vs Monte Carlo Algorithms

$$[3, 5, 1, 2, 4]$$

NOTE: MÅSKE UDSKIFT LV vs MC MED BEVIS AF MIN-CUT's KØRETID?!?!?



Presentation

Hey guys. I will be talking about Randomized Algorithms. I have here a program of the things I will go through *Hand out program*. As you can see, mine is identical to yours and it does not contain any information that it is not allowd to *Show own program*. First, I will be giving a brief introduction to the topic. Then I will be talking about quicksort. Lastly, I will be giving a brief introduction to Las Vegas algorithms and Monte Carlo algorithms.

Quicksort works by recursively picking a pivot element and then group all elements smaller than the pivot element in one set and all of the elements greater than the pivot element in another set, making the pivot element be in its sorted index. Then the algorithm is performed on the two sets recursively.

Consider running the algorithm on the provided example. If we were to always pick the "median element" as the pivot element, this would result in a balanced search tree, so we might expect the number of comparisons to be something like $n \log n$.

However, if we instead were to pick an extreme element as the pivot, this would result in an extremely unbalanced binary search tree, so we might expect the number of comparisons to be someting like n^2.

Picking the pivot element thus have a great effect on the running time of the algorithm. For this reason, we can instead choose the pivot element uniformly at random. By doing so we can expect the number of comparisons to be $O(n \log n)$, which I will prove now.

We start off by letting $S_{(i)}$ denote the ith smallest element in the sorted array S, for $1 \leq i \leq n$. Then, we define

$$X_{ij} = \begin{cases} 1 & \text{if } S_{(i)} \text{ and } S_{(j)} \text{ are compared in an execution} \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the total number of comparisons is

$$\sum_{i<j} X_{ij} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}.$$

Then, we are interested in the expected number of comparisons

$$\mathbb{E}\left[\sum_{i<j} X_{ij}\right] = \sum_{i<j} \mathbb{E}\left[X_{ij}\right].$$

Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared in an execution. Since X_{ij} only assumes the values 0 and 1 we have

$$\mathbb{E}[X_{ij}] = p_{ij} \cdot 1 + (1 - p_{ij}) \cdot 0 = p_{ij}.$$

Each recursive call returns some sublist $[S_{(a)}, ..., S_{(b)}]$. Let $x = S_{(c)}$ be the pivot. Suppose $a \leq i < j \leq b$. If $c < i$ or $c > j$, then $S_{(i)}$ and $S_{(j)}$ are not compared now, but are in the same subtree and might be compared later. If $i < c < j$, then $S_{(i)}$ and $S_{(j)}$ are never compared. If $c = i$ or $c = j$ then $S_{(i)}$ and $S_{(j)}$ are compared once, right now. Thus, $S_{(i)}$ and $S_{(j)}$ are compared iff $S_{(i)}$ or $S_{(j)}$ is first of $S_{(s)}, ..., S_{(j)}$ to be chosen as pivot. Hence why p_{ij} is the conditional probability of picking $S_{(i)}$ or $S_{(j)}$ given that the pivot is picked uniformly at random in $S_{(i)}, S_{(i+1)}, ..., S_{(j)}$

$$p_{ij} = \mathbb{P}\left[c \in \{i, j\} | c \in \{i, i+1, ..., j\}\right] = \frac{2}{|\{i, i+1, ..., j\}|} = \frac{2}{j - i + 1}.$$

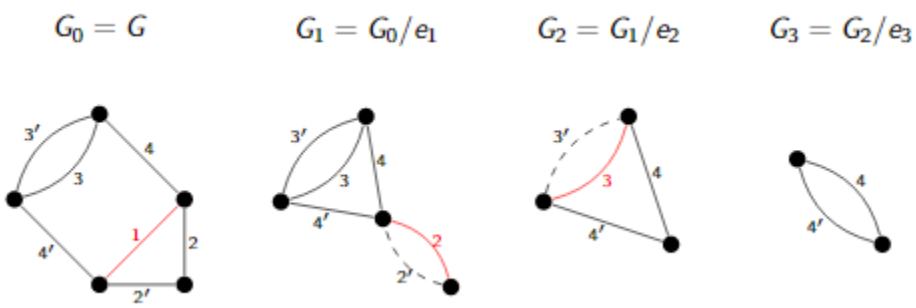
We thus have

$$
\begin{aligned}
\mathbb{E}\left[\sum_{i<j} X_{ij}\right] &= \sum_{i<j} \mathbb{E}\left[X_{ij}\right] && \text{(Linearity of expectation)} \\
&= \sum_{i<j} p_{ij} \\
&= \sum_{i<j} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} && \left(\text{Using } \sum_{i<j} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n}\right) \\
&= \sum_{i=1}^{n-1} \sum_{k=2}^{n-i+1} \frac{2}{k} \\
&< \sum_{i=1}^{n} \sum_{k=2}^{n} \frac{2}{k} && \text{(Adding more positive terms increases total value)} \\
&= 2n \sum_{k=2}^{n} \frac{1}{k} \\
&= 2n \left(\left(\sum_{k=1}^{n} \frac{1}{k}\right) - 1\right) \\
&= 2n(H_n - 1) && \left(\text{Using } H_n = \sum_{k=1}^{n} \frac{1}{k}\right) \\
&\leq 2n \int_1^n \frac{1}{x} dx = 2n \ln n = O(n \log n) && \left(\text{Using } H_n - 1 \leq \int_1^n \frac{1}{x} dx = \ln n\right)
\end{aligned}
$$

Now, onto min-cut. In min-cut we G be a connected, undirected graph with n vertices. A *cut* in G is a set of edges whose removalresults in G being broken into two or more components. The goal is to find a cut with minimal cardinality.

The algorithm works by repeatedly picking an edge uniformly at random and merging the two vertices at its end-points. Edges between vertices that are merged are removed, so that there are never any self-loops.

By running this algorithm on the example we potentially get the following Thus, the edges 4 and $4'$ is



a min-cut.

Lastly, let us talk about Las Vegas and Monte Carlo algorithms. A Las Vegas algorithm is a randomized algorithm that always return the correct output, and the randomness lies in its running time. A Monte

Carlo algorithm on the other hand is a randomized algorithm that does not necessarily return a correct output, however, we can run it multiple times to get a correct output.

Extras

Lower-bound of finding min-cut

The min-cut algorithm has a probability of $\frac{2}{n(n-1)}$ of finding a specific min-cut.
For any vertex v in a multigraph G, the *neighborhood* of v, denoted $\Gamma(v)$, is the set of vertices of G that are adjacent to v. The *degree* of v, denoted $d(v)$, is the number of edges incident on v. For a set S of vertices of G, the neighborhood of S, denoted $\Gamma(S)$, is the union of the neighborhoods of the constituent vertices.

Let k be the min-cut size and $G_i = (V_i, E_i)$ be the graph G after i iterations, which has $n_i = n - i$ vertices. We fix our attention on a particular min-cut C with k edges. Then, G_i has at least $n_i|C|/2$ edges because

$$\frac{1}{2}\sum_{v\in V_i} d_i(v) \geq \frac{1}{2}n_i|C| \quad \text{(Using } d_i(v) \geq |C|\text{)}.$$

We will bound from below the probability that no edge of C is ever contracted during an execution of the algorithm.

Let ϵ_i denote the event of not picking an edge of C at the ith step, for $1 \leq i \leq n-2$. Let $p_i = \mathbb{P}\left[\epsilon_i | \cap_{j=i}^{i-1} \epsilon_j\right]$. The probability of picking an edge of C in the ith iteration, given that no edge of C has been picked in a previous iteration, is thus

$$\begin{aligned} 1 - p_i &= \mathbb{P}\left[\epsilon_i | \cap_{j=i}^{i-1} \epsilon_j\right] \\ &= \mathbb{P}\left[e \in E_{i-1} is in C | \cap_{j=1}^{i-1} \epsilon_j\right] \\ &= \frac{|C|}{|E_{i-1}|} \leq \frac{|C|}{\frac{1}{2}n_{i-1}|C|} \qquad = \frac{2}{n_{i-1}} = \frac{2}{n-(i-1)} p_i \geq 1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1}. \qquad \Rightarrow \end{aligned}$$

Thus, the probability of finding C can then by found by

$$\begin{aligned} \mathbb{P}\left[C \text{is found}\right] &= \prod_{i=1}^{n-2} p_i \\ &\geq \prod_{i=1}^{n-2} \frac{n-1-i}{n+1-i} \\ &= \frac{n-2}{n}\frac{n-3}{n-1}\frac{n-4}{n-2}\cdots\frac{3}{5}\frac{2}{4}\frac{1}{3} \\ &= \frac{2}{n(n-1)}. \end{aligned}$$

Suppose we were to repeat the min-cut algorithm $n^2/2$ times, making independent random choices each time. The probability that a min-cut is not found in any of the $n^2/2$ attempts is at most

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

Questions they can ask

What is the probability of the min-cut algorithm succeeding?

The probability of finding a specific min-cut is greater than or equal to

$$\frac{2}{n(n-1)}.$$

If we were to repeat the algorithm x amount of times, which probability would we get for a specific x?

Suppose we were to repeat the min-cut algorithm $n^2/2$ times, making independent random choices each time. The probability that a min-cut is not found in any of the $n^2/2$ attempts is at most

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

Hashing

Disposition

1. Introduction
2. Hashing with chaining:
   (a) Introduction
   (b) Proof of expected number of elements in $L[h(x)]$
3. Signatures:
   (a) Introduction
   (b) Proof of upperbound on probability of collision

Presentation

Van Emde Boas Trees

Disposition

- Introduction
- Example on to run `Member`, `Insert` and `Successor`
- Proof of Running-time

Presentation

NP-Completeness

Disposition

1. Introduction
2. Introduction to encoding
3. Introduction to formal language
4. Definition of `P`, `NP` and `NPC` (and `NP-hard`),
5. Introduction to reducibility
6. Introduction to the ham-cycle problem
7. Introduction to the traveling-salesman problem (TSP)
8. Proof of TSP being NP-complete.

Presentation

Exact Exponential Algorithms and Parameterized Complexity

Disposition

1. Introduction
2. *Travelling salesman problem*:
   (a) Introduction with example
   (b) Naive approach + running time analysis
   (c) Dynamic programming - example of running + running time analysis
3. *Bar fight prevention*
   (a) Introduction with example
   (b) Naive approach + running time analysis
   (c) Kernelization - example of running + running time analysis
4. FPT vs XP

Presentation

Approximation Algorithms

Disposition

1. Introduction
2. Definition of the *approximation ratio*, a $\rho(n)$-*approximation algorithm* and a *randomized* $\rho(n)$-*approximation algorithm*.
3. The Vertex-cover problem
   (a) Introduction
   (b) Proof that `APPROX-VERTEX-COVER` is a 2-approximation algorithm
4. MAX-3-CNF
   (a) Introduction
   (b) Proof that the randomized algorithm for `MAX-3-CNF` is a randomized 8/7-approximation algorithm

Presentation

Definition of *approximation ratio*
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

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

Definition of $\rho(n)$-*approximation algorithm*
If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-***approximation algorithm***.

Definition of *randomized* $\rho(n)$-*approximation algorithm*
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 procuded by the randomized algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

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

We call a randomized algorithm that achieves an approximation ratio of $\rho(n)$ a ***randomized*** $\rho(n)$-***approximation algorithm***

Introduction to *vertex cover*
A ***vertex cover*** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, then either $u \in V'$ or $v \in V'$ (or both). 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 undirected graph. We call such a vertex cover an ***optimal vertex cover***.

The set C of vertices that is returned by `APPROX-VERTEX-COVER` is a vertex cover, since the laogrithm loops until every edge in $G.E$ has been covered by some vertex in C.

Algorithm 1 APPROX-VERTEX-COVER

Require: Undirected graph G
1: $C = \emptyset$
2: $E' = G.E$
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' edge (u, v) and every edge incident on either u or v
7: **return** C

Proof that `APPROX-VERTEX-COVER` is a 2-approximation algorithm
Let A denote the set of edges that line 4 picked. Not two edges in A share an endpoint. Thus no two edges in A are covered by the same vertex from an optimal cover C^*, and we have the lower bound

$$|C^*| \geq |A| \tag{1}$$

on the size of an optimal vertex cover. Since A consists of the edges between two vertices in C (and since all of the elements in C are unique), we have the (exact) upper bound on the size of the vertex cover returned

$$|C| = 2|A| \tag{2}$$

Combining equation (1) and (2), we obtain

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

Polygon Triangulation

Disposition

1. Introduction
2. The 3-coloring approach
   (a) Example on running the algorithm
   (b) Proving that the 3-coloring approach is optimal in worst case
3. Example on partitioning a polygon into monotone pieces + runtime analysis
4. Example on triangulating a monotone polygon + runtime analysis

Presentation