Advanced Algorithms and Datastructures - Exam Notes

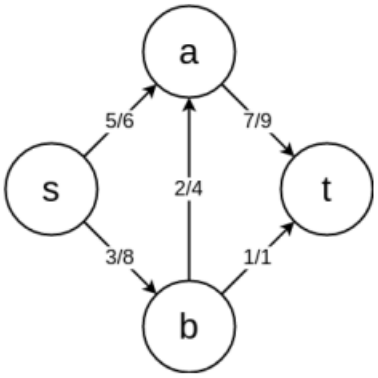
André O. Andersen

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 $f(S, T) = |f|$.

Questions they can ask

Linear Programming and Optimization

Program

1. Introduction
2. Preparing and running SIMPLEX on example
3. Proof of weak duality

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

Presentation

Hey guys. I will be talking about Max flow. 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.

In a linear-programming problem we wish to optimize a linear function subject to a set of linear inequalities. A linear-programming problem consists of an ***objective function***, that we wish to optimize, and some ***linear constraints***, which multiple non-strict inequalities that are used to restricting the solution of the objective function.

To solve a linear-programming problem, one can make use of the ***simplex algorithm***, which takes as input a linear program and returns an optimal solution.

In ***standard form*** we are given n real numbers $c_1, c_2, ..., c_n$, m real numbers $b_1, b_2, ..., b_m$, and mnreal numbers a_{ij} for $i = 1, 2, ..., m$ and $j = 1, 2, ..., n$. We then wish to find n real numbers $x_1, x_2, ..., x_n$ that maximize

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

subject to

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

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

where the last constraint is required in standard form.

To use the simplex algorithm, the linear program should be in ***slack form***, where the nonnegativity constraints are the only inequality constraints, and the remaining constraints are equalities.

When preparing a linear program, we often need to transform it into another equivalent linear program. 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 $-z$.

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

$$2x_1 - 3x_2.$$

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 $x_2', x_2'' \geq 0$. Thus, by doing so we receive the

objective function $2x_1 - 3x_2' + 3x_2''$, the two constraints

$$x_1 + x_2' + x_2'' = 7$$

and

$$x_1 - 2x_2' + 2x_2'' \leq 4,$$

and the nonnegativity constraint

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

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

$$x_1 + x_2' + x_2'' = 7$$

with the two constraints

$$x_1 + x_2' + x_2'' \leq 7$$

and

$$x_1 + x_2' + x_2'' \geq 7.$$

Lastly, we see, that the constraint

$$x_1 + x_2' + x_2'' \geq 7$$

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

$$-x_1 - x_2' - x_2'' \leq 7.$$

In total we wish to maximize the objective function

$$2x_1 - 3x_2' + 3x_2''$$

and the constraints

$$\begin{aligned} x_1 - 2x_2' + 2x_2'' &\leq 4 \\ x_1 + x_2' + x_2'' &\leq 7 \\ -x_1 - x_2' - x_2'' &\leq 7 \\ x_1, x_2', x_2'' &\geq 0. \end{aligned}$$

For consistency in variable names, we rename x_2' to x_2 and x_2'' to x_3 and obtain

$$\begin{aligned} x_1 - 2x_2 + 2x_3 &\leq 4 \\ x_1 + x_2 + x_3 &\leq 7 \\ -x_1 - x_2 - x_3 &\leq 7 \\ x_1, x_2, x_3 &\geq 0. \end{aligned}$$

and the objective function we wish to maximize

$$2x_1 - 3x_2 + 3x_3.$$

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. By doing so we get the linear programming

$$z = 2x_1 - 3x_2 + 3x_3$$

$$\begin{aligned} x_4 &= 4 - x_1 + 2x_2 - 2x_3 \\ x_5 &= 7 - x_1 - x_2 - x_3 \\ x_6 &= 7 + x_1 + x_2 + x_3 \end{aligned}$$

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 *Run SIMPLEX*.

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

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

Extras

Questions they can ask

Randomized Algorithms

Disposition

1. Introduction
2. Example on running randomized quicksort + motivation behind randomness
3. Analysis of expected runtime of randomized quicksort
4. Example on running the min-cut algorithm
5. Las Vegas Algorithms vs Monte Carlo Algorithms

Presentation

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