Exam Notes
Advanced Algorithms

Kasper Nybo Hansen
nybo@diku.dk

June 13, 2011

Contents

1 Max Flow

The problem of max flow is finding a flow in a directed graph from a source to a sink. In the case of multiple sources and sinks, one can extend the graph by introducing a *super source* and a *super sink*.

1.1 Definitions

Let $G = (V, E)$ be a directed graph containing a source vertex $s \in V$ and sink vertex $t \in V$. Let every node $v \in V$ be reachable from s and let t be reachable from every node $v \in V$. Let each edge $(u, v) \quad \forall u, v \in V$ have a capacity $c(u, v) \geq 0$. The a flow is a real-valued function $f : V \times V \leftarrow R$, satisfying the capacity constraint and the flow conservation property. The capacity constraint says that the flow of an edge cannot exceed the capacity i.e

$$0 \leq f(u, v) \leq c(u, v) \tag{1}$$

and the flow conservation property says that the total flow going into a vertex u must be equal to the total flow going out of the vertex u, where $u \in V \setminus \{s, t\}$ i.e

$$\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v) \qquad u \in V \setminus \{s, t\} \tag{2}$$

The flow value, $|f|$, is calculated as

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s) \tag{3}$$

i.e. the flow value is calculated as the total flow going out of the source vertex minus the total flow going in.

1.2 Residual network

Let f be a feasible flow in $G = (V, E)$. Let G_f denote the residual network of G induced by f, and let f' be a flow in G_f. Then the function

$$(f \uparrow f')(u, v) = f(u, v) + f'(u, v) - f'(v, u) \qquad \forall (u, v) \in E \tag{4}$$

denotes the augmentation of flow f by flow f'.

It can be shown that $f \uparrow f'$ is a flow in G and that the value is $|f \uparrow f'| = |f| + |f'|$.

1.3 Cut

Given a graph $G = (V, E)$ a cut is a partitioning such that V is divided into two subset X and Y. We define the flow across a cut as

$$f(X, Y) = \sum_{x \in X} \sum_{y \in Y} f(x, y) - \sum_{x \in X} \sum_{y \in Y} f(y, x) \tag{5}$$

thus the flow value of a cut is the total flow going from one subset to the other minus the total flow going the other way.

Let $G = (V, E)$, $X =\subset V$, $Y = V \setminus X$, $s \in X$ and $t \in Y$. The partition (X, Y) is then called a cut. The *net flow* is defined as $f(X, Y)$. The capacity of the cut is defined as

$$c(X, Y) = \sum_{u \in X, v \in Y} c(u, v) \tag{6}$$

thus the capacity of a cut, is the total capacity of all connections between the two subsets.

1.4 Min-cut theorem

A Min-cut is a cut where the capacity over all cuts is minimized. We thus choose to partition the V such that the capacity, $c(X, Y)$, between the two sets are minimized.

Show that an augmenting path increases the flow value. This is intuitively true, since when finding an augmenting path, it is added to the original graph.

Show that the flow value has an upper bound. Let V, in a graph $G = (V, E)$, be partitioned into two subsets, S and T. The flow value then has a upper bound equal to the capacity of the cut. The following proves this statement

$$\begin{aligned}
|f| &= f(S, T) \\
&= \sum_{s \in S} \sum_{t \in T} f(s, t) - \sum_{s \in S} \sum_{t \in T} f(t, s) \\
&\leq \sum_{s \in S} \sum_{t \in T} f(s, t) \\
&\leq \sum_{s \in S} \sum_{t \in T} c(s, t) \\
&= c(S, T)
\end{aligned}$$

Theorem 1.1. *Given a graph $G = (V, E)$ and a flow f (f does not necessarily be a max flow), the min-cut theorem states that the following three statements are equivalent*

1. *f is a maximum flow in G, i.e the flow value, $|f|$, is maximized*
2. *There are no augmenting paths in the residual network, G_f*
3. *$|f| = c(S, T)$ for some cut (S, T) in G*

Proof. The proof is done in a circular fashion. We will start by showing $1 \Rightarrow 2$ by contradiction.

Let f be a maximum flow in G. Let there be an augmenting path in the residual network. Hence there is a nonzero flow f' in G_f. Thus $f \uparrow f'$ is stictly greater than f. This is a contradiction compared to our initial belief that f was a maximum. □

We will now show $2 \Rightarrow 3$

Proof. Assume that there are no augmenting paths in G_f, i.e. there are no paths between s and t in G_f, and they are there disconected. Now partition G into two subsets, S and T, such that $S = \{v \in V : \text{There is a path from } s \text{ to } v \text{ in } G_f\}$ and $T = V \setminus S$. I.e. S contains the vertices reachable from the source node s in G_f. Then (S, T) is a cut. We now want to show that the value of this cut is equal to $|f|$.

Let $u \in S$ and $v \in T$.

If the edge $(u, v) \in E$, i.e. (u, v) is an edge going from S to T, then the flow going through this edge must be equal to the capacity, i.e. $f(u, v) = c(u, v)$ otherwise $(u, v) \in E_f$ which would place v in set S.

If the edge $(v, u) \in E$, i.e. (v, u) is an edge going from the T to S, then we must have that $f(v, u) = 0$, otherwise $c_f(u, v) = f(v, u)$ would be positive and therefore $(u, v) \in E_f$ which would place v in set S.

Thus the flow of the cut is

$$\begin{aligned} f(S,T) &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\ &= \sum_{u \in S} \sum_{v \in T} f(u,v) - 0 \\ &= c(S,T) \end{aligned}$$

□

The final thing we need to show is $3 \Rightarrow 1$. We thus need to show that $|f| = c(S, T)$ implies that $|f|$ is maximized.

Proof. We have shown that a flow has a upper bound of $c(S, T)$ i.e. $|f| \leq c(S, T)$. This means that $|f|$ must be maximized when $|f| = c(S, T)$. □

The min-cut is the dual of the max flow.

1.5 Ford-Fulkerson and Edmonds Karp

The Ford-Fulkersen method works as follows:

```
Initialize all edges to have a 0 flow
Construct the residual network G_f
while augmenting paths exists in G_f do
  Find an augmenting path in G_f
  augment f with the max value found in the augmenting path
  Construct the residual network for the increased flow
loop
```

An example of running the Ford-Fulkerson method can be seen on figure 1

The running time of the Ford-Fulkerson can be expressed as $\mathcal{O}(E\dot{f}*)$ where $f*$ denotes the value of the max flow. At each iteration we search through $\mathcal{O}(E)$ edges, and the maximum number of iterations is $f*$. An illustration of this running time can be seen in figure 2.

This running time can be improved by selecting the augmenting path by breath first. Breath first chooses the shortest path with capacity between the source and sink. This is exactly what Edmonds-Karp does, thus Edmonds-Karp is a specialized version of the Ford-Fulkerson method. The running time of Edmonds-Karp is $\mathcal{O}(V^2E)$.

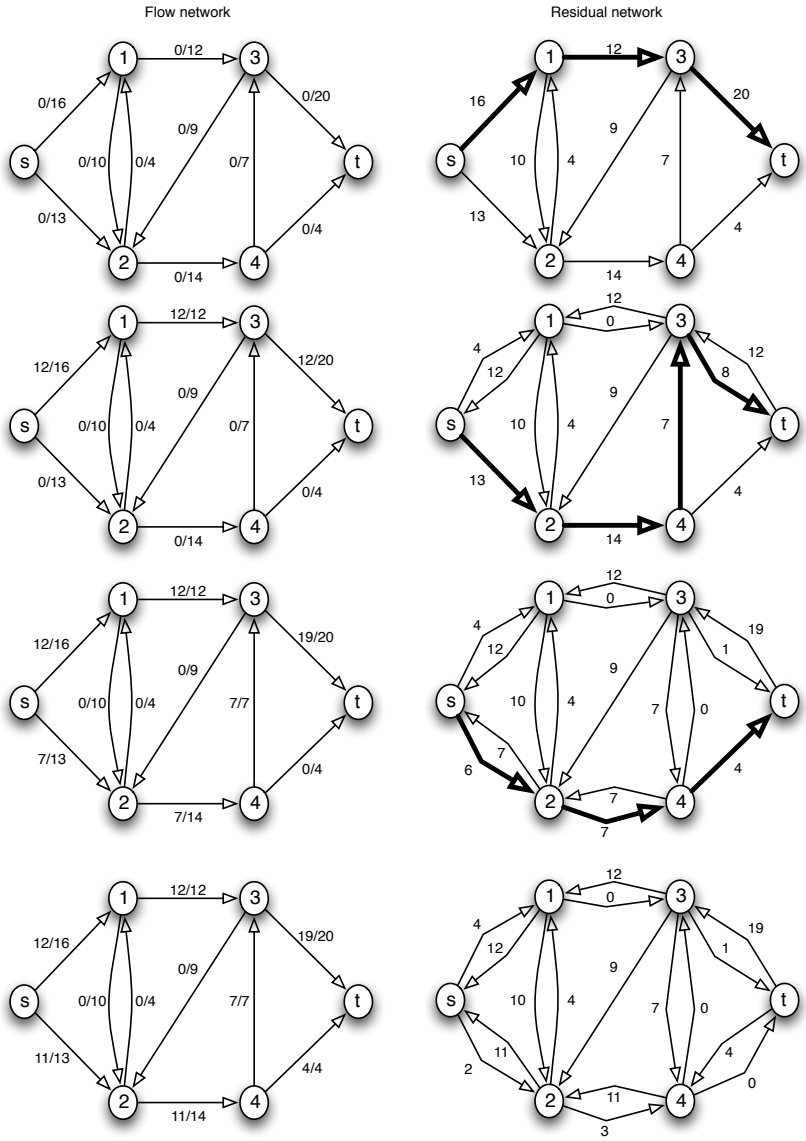

Figure 1 – Example of running the Ford-Fulkerson method. Left side is the flow network, right side is the residual network, with bold lines representing augmenting paths.

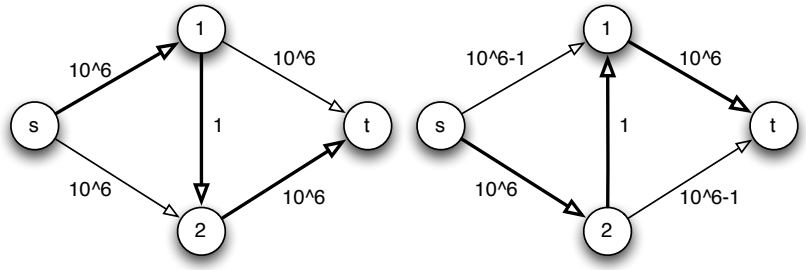

Figure 2 – Illustration of the complexity of the Ford-Fulkerson algorithm

2 Linear Programming

In linear programming we want to maximize (or minimize) a linear function, under a number of constraints. An example of a linear program is

$$\begin{aligned} \max \quad & x_1 + x_2 \\ \text{St.} \quad & 4x_1 - x_2 \leq 8 \\ & 2x_1 + x_2 \leq 10 \\ & 5x_1 - 2x_2 \leq -2 \\ & x_1, x_2, x_3 \geq 0 \end{aligned} \tag{7}$$

7 is called the objective function.

A linear program in 2 dimensions can be illustrated as figure 3. The gray area is called the feasible region.

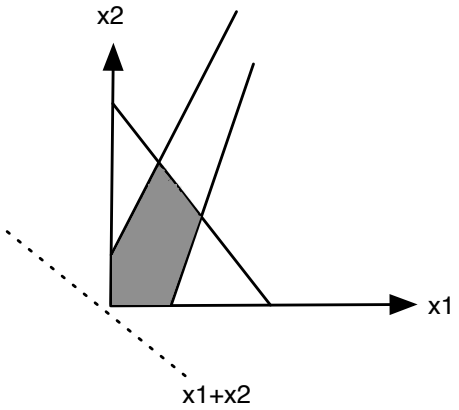

Figure 3 – Illustration of a simple linear program where the objective function is $x_1 + x_2$. The feasible region is gray.

A linear program can be written in two ways. Standard form and slack form. When a linear program is written in standard form it takes the following form

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{S.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad i = 1, 2, \ldots, m \\ & x_j \geq 0 \qquad i = 1, 2, \ldots, n \end{aligned}$$

where m is the number of constraints and n is the number of variables. One could think of a as a matrix having, m rows and a columns. Every linear program can be rewritten to slack form.

2.1 Converting into standard form

A linear program does not have to take standard form. But we can always turn a linear program into standard form. The following section describes how to convert a non-standard form linear program into standard form.

- If the program is a minimization problem, then it can be turned into a maximization problem by multiplying the objective function by -1. The constraints are not changed, since the feasible region is the same, regardless if we want to maximize or minimize an objective function under the same constraints.
- If the linear program contains variables, x_j without nonnegative constraints we can replace x_j so $x_j = x'_j - x''_j$, and add the constraints $x'_j, x''_j > 0$.
- If the linear program contains equality constraints, then replace each equality with to inequality constraints going opposite ways, i.e $a = b \Leftrightarrow a \leq b \wedge a \geq b$.
- If the linear program contains inequality constraints, but instead of less than, they are greater than, then multiply the constraint with -1.

The following is an example of how to convert a non-linear program into standard form.

$$\begin{aligned} \min \quad & -2x_1 + 3x_2 \\ \text{St.} \quad & x_1 + x_2 = 7 \\ & x_1 - 2x_2 \leq 4 \\ & x_1 \geq 0 \end{aligned}$$

Start by making it a maximization problem, i.e. multiply the objective function with -1.

$$\begin{aligned} \max \quad & 2x_1 - 3x_2 \\ \text{St.} \quad & x_1 + x_2 = 7 \\ & x_1 - 2x_2 \leq 4 \\ & x_1 \geq 0 \end{aligned}$$

Next place x_2 under the nonnegative constraint. Let $x_2 = x'_2 - x''_2$ then

$$\begin{aligned} \max \quad & 2x_1 - 3x'_2 + 3x''_2 \\ \text{St.} \quad & x_1 + x'_2 - x''_2 = 7 \\ & x_1 - 2x'_2 + 2x''_2 \leq 4 \\ & x_1, x'_2, x''_2 \geq 0 \end{aligned}$$

Make a cleanup so we remove those pesky primes, just rename $x_2' = x_2$ and $x_2'' = x_3$.

$$\begin{aligned} \max \quad & 2x_1 - 3x_2 + 3x_3 \\ \text{St.} \quad & x_1 + x_2 - x_3 = 7 \\ & x_1 - 2x_2 + 2x_3 \leq 4 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Lets remove the equality sign, by replacing the equality with to opposite directed inequalities i.e.

$$\begin{aligned} \max \quad & 2x_1 - 3x_2 + 3x_3 \\ \text{St.} \quad & x_1 + x_2 - x_3 \leq 7 \\ & x_1 + x_2 - x_3 \geq 7 \\ & x_1 - 2x_2 + 2x_3 \leq 4 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

Final thing we need to do, is to turn the greater than into a less than my multiplying with -1, i.e.

$$\begin{aligned} \max \quad & 2x_1 - 3x_2 + 3x_3 && (8) \\ \text{St.} \quad & x_1 + x_2 - x_3 \leq 7 \\ & -x_1 - x_2 + x_3 \leq -7 \\ & x_1 - 2x_2 + 2x_3 \leq 4 \\ & x_1, x_2, x_3 \geq 0 && (9) \end{aligned}$$

and the linear program in now in standard form.

2.2 Slack form

In slack form, the inequalities are replaced by equalities and a number of slack variables are introduced. We can rewrite the standard form linear program shown in 8 to slack form as

$$\begin{aligned} \max \quad & 2x_1 - 3x_2 + 3x_3 \\ \text{St.} \quad & x_4 = 7 - x_1 - x_2 + x_3 \\ & x_5 = -7 + x_1 + x_2 - x_3 \\ & x_6 = 4 - x_1 + 2x_2 - 2x_3 \\ & x_1, x_2, x_3, x_4, x_5, x_6 \geq 0 \end{aligned}$$

We have just introduced the three *basic variables* x_4, x_5, x_6 and rearranged the original non-basic variables. In general when going from standard to slack form, you need m new slack variables, where m denotes the number of constraints.

2.3 SIMPLEX algorithm

There exists several methods to solve linear programs. Some methods are polynomial, but doesn't work well in practice. A non polynomial method is called simplex. Simplex has it name from the shape of the feasible region.

The geometric interpretation of SIMPLEX is as follows. It starts at a corner point: the origin, and looks at the edges incident with that point. Then it chooses one, and moves along an edge which makes the objective value get larger (or stay the same). If the edge goes on forever, then the LP is unbounded; if the edge does not go on forever, it ends up at another corner point. Then the Simplex Method does the same thing over and over again, until it stops at a certain point. That point is the place where the objective value is the largest.

Normally, this would only lead you to a local maximum, instead of the global maximum, but this cannot happen when solving a LP. This is because the objective values and inequalities are linear in form.

Given a linear program

$$\begin{aligned} \max \quad & 2x_1 - x_2 && (10) \\ \text{St.} \quad & 2x_1 - x_2 \leq 7 \\ & x_1 - 5x_2 \leq -7 \\ & x_1, x_2 \geq 0 && (11) \end{aligned}$$

Simplex needs the problem in standard form, it starts by converting the problem into slack form, i.e.

$$\begin{aligned} z &= 2x_1 - x_2 && (12) \\ x_3 &= 7 - 2x_1 + x_2 \\ x_4 &= -7 - x_1 + 5x_2 \\ x_1, x_2, x_3, x_4 &\geq 0 && (13) \end{aligned}$$

it then finds a initial solution. Typically this solution is found by setting the variables to 0. Sometimes the initial solution isn't feasible as can be see in 12. In this case, we need to formulate an auxiliary program. The auxiliary program for 10 is

$$\begin{aligned} \max \quad & -x_0 && (14) \\ \text{St.} \quad & 2x_1 - x_2 - x_0 \leq 2 \\ & x_1 - 5x_2 - x_0 \leq -4 \\ & x_1, x_2, x_0 \geq 0 \end{aligned}$$

The auxiliary program is then solved with the SIMPLEX algorithm. It starts

by rewriting the problem into slack form

$$\begin{aligned} z &= -x_0 \\ x_3 &= 2 - 2x_1 + x_2 + x_0 \\ x_4 &= -4 - x_1 + 5x_2 + x_0 \\ x_0, x_1, x_2, x_3, x_4 &\geq 0 \end{aligned} \tag{15}$$

And pivot's, so x_0 enters the basis, on the constraint that is most negative

$$\begin{aligned} z &= -x_0 \\ x_3 &= 2 - 2x_1 + x_2 + x_0 \\ x_0 &= 4 + x_1 - 5x_2 + x_4 \\ x_0, x_1, x_2, x_3, x_4 &\geq 0 \end{aligned} \tag{16}$$

and substitutes x_0 into the objective function and the constraints

$$\begin{aligned} z &= -4 - x_1 + 5x_2 - x_4 \\ x_0 &= 4 + x_1 - 5x_2 + x_4 \\ x_3 &= 6 - x_1 - 4x_2 + x_4 \\ x_0, x_1, x_2, x_3, x_4 &\geq 0 \end{aligned} \tag{17}$$

The basic feasible solution to this problem is $(x_0, x_1, x_2, x_3, x_4) = (4, 0, 0, 6, 9)$. We want the optimal solution so we need to do a pivot. Since $5x_2$ is the only one that is nonnegative, we investigate this: x_0 binds with $4/5 = 16/20$, and x_3 binds with $6/4 = 30/20$. We thus choose to pivot on x_0 and x_2, since x_0 is the most binding.

$$\begin{aligned} z &= -4 - x_1 + 5(\frac{4}{5} - \frac{1}{5}x_0 + \frac{1}{5}x_1 + \frac{1}{5}x_4) - x_4 \\ x_2 &= \frac{4}{5} - \frac{1}{5}x_0 + \frac{1}{5}x_1 + \frac{1}{5}x_4 \\ x_3 &= 6 - x_1 - 4(\frac{4}{5} - \frac{1}{5}x_0 + \frac{1}{5}x_1 + \frac{1}{5}x_4) + x_4 \\ x_0, x_1, x_2, x_3, x_4 &\geq 0 \end{aligned} \tag{18}$$

reducing yields

$$\begin{aligned} z &= -x_0 \\ x_2 &= \frac{4}{5} - \frac{1}{5}x_0 + \frac{1}{5}x_1 + \frac{1}{5}x_4 \\ x_3 &= \frac{14}{5} + \frac{4}{5}x_0 - \frac{9}{5}x_1 + \frac{1}{5}x_4 \\ x_0, x_1, x_2, x_3, x_4 &\geq 0 \end{aligned} \tag{19}$$

this slack form is the final solution to the auxiliary problem. Since $x_0 = 0$ we know it's optimal, and that the initial problem was feasible. Since $x_0 = 0$ we

can remove it from the set of constraints. The original objective function 12 can then be restored, substituting the basic variables from 10, and retaining the constraints form 19

$$\begin{aligned} z &= -\frac{4}{5} + \frac{9}{5}x_1 - \frac{1}{5}x_4 \\ x_2 &= \frac{4}{5} + \frac{1}{5}x_1 + \frac{1}{5}x_4 \\ x_3 &= \frac{14}{5} - \frac{9}{5}x_1 + \frac{1}{5}x_4 \\ x_1, x_2, x_3, x_4 &\geq 0 \end{aligned} \tag{20}$$

Setting this slack form to zero, yields the basic solution $(x_1, x_2, x_3, x_4) = (0, \frac{4}{5}, \frac{14}{5}, 0)$. This slack form can now be solved by using the SIMPLEX algorithm again.

a flow chart of the simplex can be seen in figure 4

2.3.1 Cycling

There is a risk that SIMPLEX cycles. This means that we at some point arrive at the same slack state as we have visited before. Cycling can be avoided by always choosing the entering variable with the smallest index, and furthermore if there are multiple exiting variables, always choose the one with the lowest index.

2.4 Duality

Duality means that finding a solution in one problem

Given a linear program with n variables and m constraints

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} c_j x_j \\ \text{S.t.} \quad & \sum_{j=1}^{n} a_{ij} x_j \leq b_i \qquad i = 1, 2, \ldots, m \\ & x_j \geq 0 \qquad i = 1, 2, \ldots, n \end{aligned} \tag{21}$$

and call this the *primal*. We define the *dual* as

$$\begin{aligned} \min \quad & \sum_{i=1}^{m} b_i y_i \\ \text{S.t.} \quad & \sum_{j=1}^{m} a_{ij} y_i \geq c_j \qquad j = 1, 2, \ldots, n \\ & y_j \geq 0 \qquad i = 1, 2, \ldots, m \end{aligned} \tag{22}$$

The dual is a minimization problem and it has m variables and n constraints. Furthermore the less-than-equal-to has been replaced by greater-than-equal-to. Each b_i in the primal is now part of the objective function as a coefficient. Each c_j in the primal is now the right handside of the constraints. If a was a matrix then in the dual it is transposed.

Example: Let the primal be

$$\begin{aligned} \max \quad & 3x_1 + x_2 + 2x_3 \\ \text{St.} \quad & x_1 + x_2 + 3x_3 \leq 30 \\ & 2x_1 + 2x_2 + 5x_3 \leq 24 \\ & 4x_1 + x_2 + 2x_3 \leq 36 \\ & x_1, x_2, x_3 \geq 0 \end{aligned} \tag{23}$$

then the dual is

$$\begin{aligned} \min \quad & 30y_1 + 24y_2 + 36y_3 \\ \text{St.} \quad & y_1 + 2y_2 + 4y_3 \leq 3 \\ & y_1 + 2y_2 + 1y_3 \leq 1 \\ & 3y_1 + 5y_2 + 2y_3 \leq 2 \\ & x_1, x_2, x_3 \geq 0 \end{aligned} \tag{24}$$

Weak linear-programming duality states the following

Theorem 2.1. *Given a feasible solution to the primal problem, $\bar{x}$, and a feasible solution to the dual problem, $\bar{y}$, then*

$$\sum_{j=1}^{n} c_j x_j \leq \sum_{i=1}^{m} b_i y_i \tag{25}$$

said in words: the primal is bounded above by the dual.

Proof. Let $\bar{x}$ and $\bar{y}$, be feasible solutions to the primal and the dual respectively.

From 22 we know that $\sum_{j=1}^{m} a_{ij} y_i \geq c_j$. Thus we can write

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{j=1}^{n} \left(\sum_{j=1}^{m} a_{ij} \bar{y}_i \right) \bar{x}_j \tag{26}$$

exchanging $\bar{x}_i$ and $\bar{y}_i$ yields

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{j=1}^{n} \left(\sum_{j=1}^{m} a_{ij} \bar{x}_j \right) \bar{y}_i \tag{27}$$

and 21 gives us $\sum_{j=1}^{m} a_{ij} x_j \leq b_i$ which we can insert

$$\sum_{j=1}^{n} c_j \bar{x}_j \leq \sum_{j=1}^{m} b_i \bar{y}_i \tag{28}$$

and we arrive what we wanted to prove. □

The duality theorem states that if a feasible solution of the primal, is equal to a feasible solution to the dual, the this feasible solution is optimal to both the primal and the dual. More precisely the theorem states

Theorem 2.2. *Let SIMPLEX terminate with a feasible basic solution* $\bar{x}* = (x_1*, x_2*, \ldots, x_n*)$*, and let* N *denote the nonbasic and* B *the basic variables for the final slack form. Let* c' *denote the coefficients of the objective function in the final slack form.*

Not let $\bar{y}* = (y_1*, y_2, \ldots, y_m)$ *be defined as*

$$\bar{y}* = \begin{cases} -c'_{n+i} & \text{if } (n+i) \in N \\ 0 & \text{otherwise} \end{cases} \tag{29}$$

then $\bar{x}*$ *is an optimal solution to the primal, and* $\bar{y}*$ *is an optimal solution to the dual and*

$$\sum_{j=1}^{n} c_j x_j* = \sum_{i=1}^{m} b_i y_i* \tag{30}$$

Proof. In order to prove the above, we have to show that $\bar{y}*$ is a feasible solution to the dual, and that

$$\sum_{j=1}^{n} c_j x_j* = \sum_{i=1}^{m} b_i y_i* \tag{31}$$

□

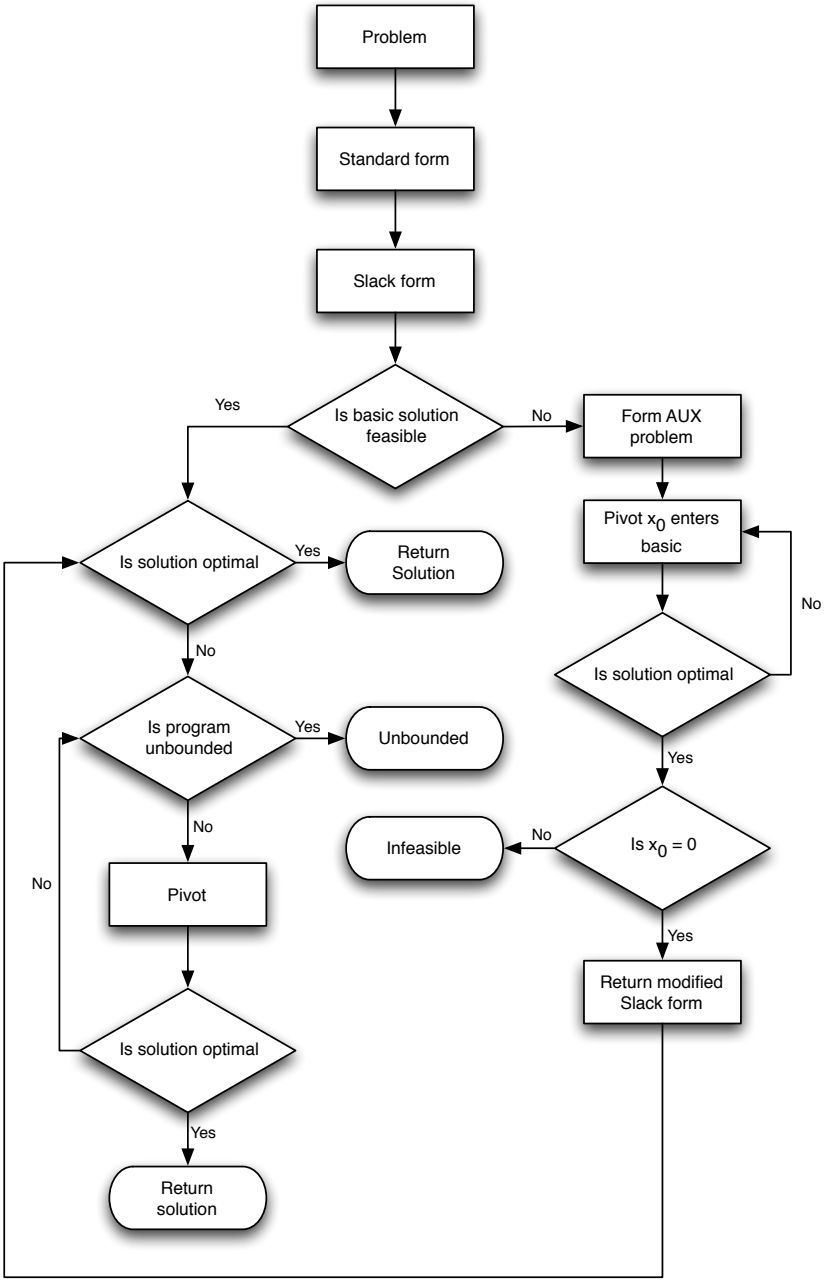

Figure 4 – Flowchart of the SIMPLEX algorithm

3 NP-completeness

NP-complete problems is a class of problems that has algorithms that run in *super polynomial* time, i.e. there running time is thus $> \mathcal{O}(N^k)$, for some constant k.

NP-completeness only applies to decision problems. A optimization problem can easily be made into a decision problem by setting a bound. E.g. a shortest path problem[1] can be made into a decision problem, by asking: Is there a shortest path between these two nodes with at most length k?.

When we are proving something is NP-complete, and therefore hard to solve, we are proving it under the constraint that no polynomial algorithm exist for a NP-complete problem. This assumption is still not proved formally (But if this does not hold, we have a BIG problem!).

3.1 Language

A language describes a problem. E.g. the language for the Hamilton cycle problem can be described as "Does the graph G have a Hamilton cycle", or more formally

$$HAM - CYCLE = \{\langle G \rangle \; G \text{ is a hamilton cycle}\} \tag{32}$$

The contents between the $\langle$ and $\rangle$ is the input to the problem.

3.2 Reduction algorithm

Suppose we have a decision problem A that we want to solve in polynomial time. Now suppose we already know how to solve a different decision problem, B, in polynomial time. Finally suppose we have a algorithm that transforms an instance $a \in A$, into an instance $b \in B$ with the following characteristics

- The transformation is done in polynomial time
- The answers are the same, i.e $a \Leftrightarrow b$.

this algorithm is then called a *reduction algorithm.*

The key point in the reduction algorithm, is that it runs in polynomial time. Suppose we had an instance of a non-polynomial problem we could transform, in polynomial time, into an other problem that could be solved by a polynomial algorithm. The final result would be a running time of $\mathcal{O}(n^{k_1}) + O(n^{k_2}) = O(n^k)$ yielding a polynomial algorithm to a non-polynomial problem.

Formally we say that a problem L_1 is polynomial time reducible to L_2, i.e. $L_1 \leq_p L2$ if there exist a function f such

$$x \in L_1 \Leftrightarrow f(x) \in L_2 \tag{33}$$

we call f a *reduction function* and it can be computed by a reduction algorithm.

[1] Finding the shortest path between two nodes in a graph

3.3 Definition of classes

We call the class of problems for which a polynomial time algorithm is known for P. More formally

Definition 3.1. *Let $p \in P$ be a problem in P. Then there exist a polynomial algorithm that can solve the p in time $\mathcal{O}(n^k)$, where n is the input size and k is a constant. The language for P is*

$$P = \{L\text{: } L \textit{ is accepted be a polynomial-time algorithm}\} \tag{34}$$

Examples of problems contained within P is most sorting algorithms, some graph algorithms etc.

We call the class of problems for which a solution can be verified in polynomial time for NP. More formally the complexity class NP has the language

$$NP = \{\text{L: There exists an A, such that L is verified in polynomial time}\} \tag{35}$$

i.e. the NP class consist of all the problem instances L for which we can verify a solution using A in polynomial time.

A problem is in $co - NP$ if it's compliment is in NP. The complement of a decision problem, is the negation of the decision problem (interchange the yes/no).

A language L is np-complete if

1. $L \in NP$
2. $L' \leq_p L \qquad \forall L' \in NP$

In the case where only property 2 is satisfied the problem is called *np-hard*.

Figure 5 illustrates the 3 classes in a set manner.

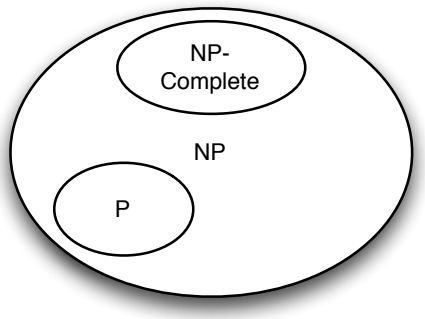

Figure 5 – Set illustration of the 3 classes of problems. Note that both P and NPC is in NP, but they are disjoint i.e. this illustration assumes that $P \neq NP$

Theorem 3.2. *Let L_1 and L_2 be languages such that $L_1 \leq_p L_2$. Then if $L_2 \in P$ then $L_1 \in P$*

Proof. We know that L_2 can be solved in polynomial time. From 33 we know that there exist a function f such that $x \in L_1 \Leftrightarrow f(x) \in L_2$. For any $x \in L_1$ we can thus use the result $f(x) in L_2$ and find the answer to the instance x. L_1 can thus be solved in polynomial time. □

Theorem 3.3. *If any NP-complete problem is solvable in polynomial time, all NP-complete problems are solvable in polynomial time.*

Proof. Suppose that $L \in P$ and also that $L \in NPC$. For any $L' \in NP$ we have $L' \leq_p L$, by property 2 of NP-completeness. Since $L \in P$ there must exist a polynomial algorithm. And by 3.2 this implies that $L' \in P$. □

3.4 Example $HAM - CYCLE \leq_p TSP$

In this example we wish to prove that it is very unlikely that a polynomial algorithm exist for the Traveling Salesman Problem (TSP). In fact, we wish to show that solving the TSP is at least as hard as solving the Hamiltonian cycle. We know that the Hamiltonian cycle problem is NP-complete.

Recall that a Hamilton cycle is a path in a undirected graph where each vertex is visited exactly once.

Thus, if we can show that it is at least as has to solve TSP as to solve the Hamiltopnian cycle, we have shown that TSP is in the NP-completeness class.

Let the language of the TSP be defined as

$$\begin{aligned} TSP = \{\langle G, c, k\rangle : & G = (V, E) \text{ is a complete graph,} \\ & c : V \times V \to \mathbb{Z}, \\ & k \in \mathbb{Z}, \\ & G \text{ has a TSP tour of at most cost } k\} \end{aligned}$$

We need to show two things. First we need to show that the TSP decision problem is in the NP class. This can be shown by showing that TSP has a verification algorithm, that can verify an instance in polynomial time.

Let the verification algorithm check the sequence of vertices and sums the total cost and verifies this as being $\leq k$. Such a verification algorithm definitely exist, and can be run in polynomial time.

The next thing we need to show is $HAM - CYCLE \leq_p TSP$.

Let $G = (V, E)$ be an instance of $HAM - CYCLE$. We can construct an instance of TSP as follows. Form the complete graph[2] $G' = (V, E')$. Let the vertices in V be the cities in the TSP. Let the cost function, $c(i, j)$ be defined by

$$c(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E \\ 1 & \text{if } (i, j) \notin E \end{cases} \tag{36}$$

So if a edge resides in the Hamilton graph G, then it has cost 0 in G' otherwise 1. An instance of TSP is thus a cycle in G' with cost at most 0.

[2] A complete graph has a path from every vertex to all other vertices.

It is clear that we can construct G' in polynomial time. We now show that G contains a Hamilton cycle if and only if graph G' has a tour of cost 0. Suppose G has a Hamilton cycle h. Each edge in h resides in G thus having cost 0 in G'. Conversely, suppose G' has a tour h' of cost 0, since the cost of the edges in G' is 0 and 1, h' can only occur if it traverses the edges of G' with 0 cost, thus h' contains only edges from E.

We have now proven that finding

4 Branch and Bound and Metaheuristics

4.1 Motivation

NP-hard problems can be solved to closed form by using an exact algorithm like branch and bound. They can be solved heuristically by using meta algorithms.

The search space for NP-hard problems are extremely large, and solving problems exact means, to some extent, that we need to traverse the whole search space. Solving heuristically means to traverse part of the search space, thereby gaining a speedup.

By using branch and bound we are trying not to visit the whole solution space, but instead only visit the solution space where we know an optimal solution could reside. Suppose we have a problem, with tree binary variables x_1, x_2, x_3. Figure 6 shows a total enumeration tree for this problem. There are in total $2^3 = 8$ solutions, and in general we can describe the number of solutions to such a problem as 2^n.

This is an example of a problem where we do not want to explore the whole search tree before we find a solution. Imagine we could prune some branches of the tree, thereby making the search tree smaller. This is exactly what Branch & Bound does.

The nodes of a Branch & Bound tree corresponds to partial solutions, i.e solutions where some part of the variables are defined and some part is not. The leafs corresponds to solutions where all the variables are defined.

A bounding function takes partial solution, and from this calculates a bound on the complete solution containing this partial solution.

The incumbent is the best known feasible solution found so far. One wishes to find an incumbent as faast as possiblle, as this means that, hopefully, some parts of the tree will have a bound worse than the incumbent, thereby giving us the possibility to discard these parts.

By the help of the bounding function, we can calculate the a bound of a node containing a partial solution. If this bound is worse than the incumbent, then we can discard this part of the tree, thereby remove parts of the search space.

4.2 Branching

The branch part of the Branch and Bound algorithm divides the problem into subproblems

In order to illustrate branching we can use the problem of finding the independent set in a graph.

Given a graph $G = (V, E)$ the independent set is the set of vertices $I \subset S$, such that for every vertex in I none of it's neighbors in V are in I. I.e. each edge in E has at most one endpoint in I. The maximal independent set, is a set independent set where $|I|$ is maximized.

A trivial algorithm to find such a set, is to construct all possible solutions by permutation, and check if it is a independent set. The running time can be calculated from the following observation. Each vertex in the graph can either

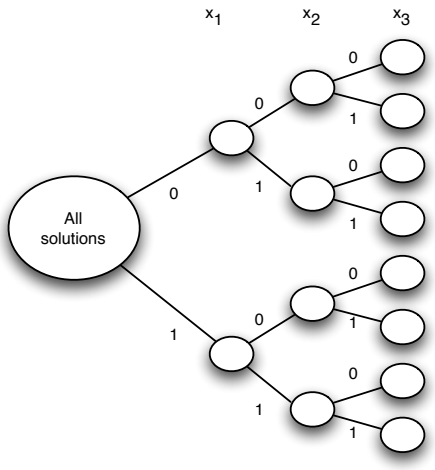

Figure 6 – Total enumeration of solutions, for a problem with tree binary variables x_1, x_2, x_3. The leafs of the tree corresponds to solutions, whereas the node is partial solutions

be included in the set or not included in the set. For each vertex we thus have 2 possibilities. Given n vertices this yields a running time of $\mathcal{O}(2^n)$!

Another approach is to use a branching algorithm. The branching algorithm works as follows. Given a graph $G = (V, E)$ pick a vertex, v with minimum degree[3]. Let the set of neighbors to v be defined as $N[v]$. For each vertex $j \in N[V]$ do a branch on the set $G \setminus N[j]$.

The running time of this procedure can be calculated as the number of nodes in the search tree T. The algorithm picks the vertex with the minimum degree and branches on itself and it's neighbors. We thus have

$$T(n) \leq 1 + T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(v_i) - 1) \tag{37}$$

where $d(v_i)$ is the degree of the $i'th$ neighboring vertex.

Since the algorithm always chooses the vertex with minimum degree we have

[3] A degree of a vertex v is the number of adjacent edges to v

$T(n-d(v)-1) \geq T(n-d(v_i)-1)$. so

$$T(n) \leq 1 + T(n-d(v)-1) + \sum_{i=1}^{d(v)} T(n-d(v_i)-1) \tag{38}$$

$$\leq 1 + T(n-d(v)-1) + \sum_{i=1}^{d(v)} T(n-d(v)-1) \tag{39}$$

$$= 1 + \sum_{i=1}^{d(v)+1} T(n-d(v)-1) \tag{40}$$

$$= 1 + (d(v)+1)T(n-d(v)-1) \tag{41}$$

now let $s = d(v)+1$ then we have

$$t(n) \leq 1 + sT(n-s) \tag{42}$$

expanding this recurrence yields

$$t(n) \leq 1 + sT(n-s) \leq 1 + s + s^2 + s^3 + \ldots + s^{n/s} \tag{43}$$

this follows from the observation, that we subtract s from n at each recursive step, so the maximal number of recursive steps is n/s. 43 is a harmonic series and can be written as $\mathcal{O}(s^{n/s})$.

4.3 Bounding

The bounding part of the Branch and Bound algorithm gives a bound of the partial solution found so far. In order to illustrate the bounding part, I have chosen to use the TSP problem.

Given a TSP problem let the root node be the fully connected graph. We can calculate a lower bound of the TSP tour by the following procedure. Choose a special node #1 and remove all it's incident edges. Calculate the minimum spanning tree, t_{rest} of the rest of the graph. Add the two shortest edges e_1, e_2 incident to #1 t_{rest} producing t_{one}. t_{one} is now a 1-tree.

Definition 4.1. *The total cost of t_{one} is less than the total cost of the optimal tour.*

Proof. Note that a Hamilton cycle in G consist of two edges e_1', e_2' and a tree t_{rest}' in the rest of G. So the set of Hamilton cycles is a subset of the set of 1-trees. Since e_1 and e_2 are the shortest edges incident to #1, and t_{rest} is the minimum spanning tree in the rest of G, the cost of t_{one} is less than or equal to the cost of any Hamilton cycle, especially a TSP tour. □

The root node of the Branch and Bound search tree of the TSP is the fully connected graph. Every time we branch on a node j, we remove edges from G so they are not present in the descendants of j. We branch on all the nodes which has degree ≥ 3.

How to get an initial Incumbent brings us to meta-heuristics

4.4 Meta-Heuristics

Meta-heuristic algorithms is algorithms that try to find a optimal solution, by improving a previous solution. Most heuristic methods starts with an initial solution, and permutes it slightly thereby giving a new, possible better, solution.

The difference between heuristic algorithms and approximation algorithms is that the approximation algorithms gives a guarantee of the result, whereas the heuristic algorithms does not give this guarantee.

All the algorithms presented below, needs an initial solution. It really depends on the problem how this initial solution is found. For the TSP problem the nearest neighbor algorithm could be used or alternatively calculating the convex hull and adding points to the edges of the hull.

Furthermore, the description of the neighborhood is very problem specific.

4.4.1 Hill climbing

Hill climbing is a simple heuristic algorithm that only makes strictly improving moves. Given a initial solution s, a sample is chosen from the neighborhood, $t \in N(s)$. If the sample yields a better solution, then the initial solution is exchanged with the sample i.e $s = t$. This procedure is repeated until local optima is reached, i.e.

$$f(t) \leq f(s) \qquad \forall t \in N(S) \tag{44}$$

The problem with Hill climbing is that local optima doesn't guarantee to bee close to the global optima, actually it could be far away from the global optimum.

4.4.2 Simulated annealing

Simulated annealing is a heuristic algorithm that always performs improving moves, but also accept non improving moves. The algorithm has a tweaking parameter T that is gradually decreased as the algorithm runs. Given a initial solution, s and a initial tweaking parameter T the algorithm chooses a sample, t from the neighborhood. If the sample yields a better solution, the initial best solution is discarded and replaced by the sample. If the sample does not yield a better solution, the sample is accepted with probability

$$e^{\frac{s-t}{T}} \tag{45}$$

4.4.3 Tabu search

Tabu search is a heuristic algorithm that has memory. It memorizes the places it has already visited, and forbids the algorithm to visit these places again. This has the effect that no cycling can occur, and no solution is explored twice.

Otherwise the algorithm works as Hill climbing, where the best solution is replaced if the candidate solution is better.

5 Approximation Algorithms

5.1 Performance ratio

The performance ratio, $\rho(n)$ is defined as

$$max(\frac{C}{C^*}, \frac{C^*}{C}) \leq \rho(n) \tag{46}$$

where C^* is the cost of the optimal overall solution, C is the cost of the solution found by the approximation algorithm and n is the number of elements. The definition of the ratio applies both to maximization and minimization problems.

In a maximization scenario typically $0 < C \leq C^*$, and C^*/C gives the factor by which the cost of C^* is greater than the cost of C. In a minimization scenario $0 < C^* \leq C$ and C/C^* tells by which factor C is greater than C^*.

If an algorithm archives an ratio of $\rho(n)$ then we call the algorithm a $\rho(n)$-approximation algorithm.

As can be seen from the definition, a ratio is equal to 1 when the cost of the optimal and the approximation is equal. This scenario corresponds to the case where the approximation equals the optimal solution.

Example: Imagine an approximation algorithm that has a worst case of twice the optimal solution disregarding the input size n. Then the approximation factor would be $\rho(n) = 2$.

5.2 The vertex cover problem

A vertex cover of an undirected graph $G = (V, E)$ is the problem of finding the subset of vertices $V' \subset V$ such that $\forall(u, v) \in E : u \in V' \vee v \in V'$, i.e. each edge is connected to a vertex in V'. The size of the vertex cover is the number of vertices in V'. The vertex cover problem is finding a vertex cover of minimum size. The vertex cover problem is NP-complete.

The approximation algorithm for vertex cover is a polynomial algorithm and works as follows: Let the input to the algorithm be a undirected graph $G = (V, E)$. Let $E' = E$ be the edges of the graph. While there are edges in E' pick a edge (u, v) from E'. Add the vertices u and v to the final vertex cover set C. Remove from E' every edge incident to either u or v.

An example run can be seen in figure 7

Figure 8 shows the optimal solution

The approximated vertex cover algorithm is a polynomial time 2-approximation (It is a polynomial time algorithm and the approximation ratio is 2.).

Proof. Let A be the set of edges that is picked in the approximation algorithm. Let $C*$ be the optimal vertex cover. Let C be the vertex cover generated by the approximation algorithm.

We can put a lower bound on C^* as follows. The number of elements in C^* must be at least $|A|$. This follows from the following observation: Each edge in A must have one of it's vertices in C^*, otherwise we would have a edge not

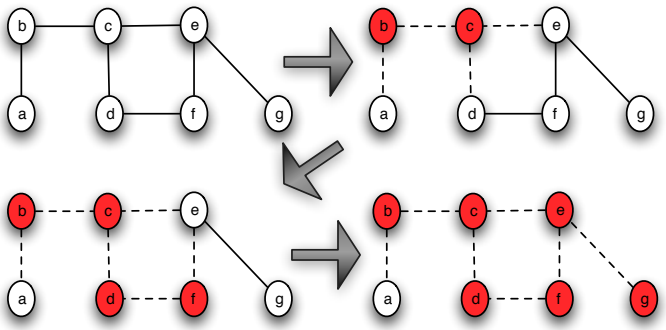

Figure 7 – Example run of the approximated vertex cover algorithm. The value is 6.

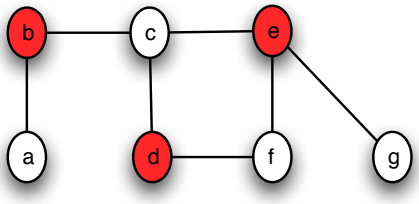

Figure 8 – Optimal solution to the vertex cover problem. The value is 3.

in C^*, and therefore an edge not part of the vertex cover which contradicts our assumption that C^* was a optimal vertex cover. Thus $|C^*| \geq |A|$. For convenience, we will rewrite this lower bound as

$$|C^*| \geq |A| \Leftrightarrow 2|C^*| \geq 2|A| \tag{47}$$

Every time the approximation algorithm picks an edge from A, it adds two vertices to C. Therefore

$$|C| = |2A| \tag{48}$$

Substituting 48 in 47 yields

$$2|C^*| \geq |C| \tag{49}$$

thereby giving us a upper-bound on the number of elements in the vertex cover found by the approximation algorithm. □

5.3 The TSP problem

In the following we assume that the triangle inequality[4] holds. If the triangle inequality do not hold, then it is possible to show that no polynomial approximation algorithm exists!

[4]Intuitively explanation: The shortest distance between 2 points is always a straight line

The TSP problem can be solved by a 2-approximation polynomial algorithm. The algorithm works as follows. It takes a bunch of points, and makes a complete graph. It then runs Prim's algoritm to find the minimum spanning tree[5]

After finding the minimum spanning tree we have a list of vertices contained in a binary heap. We then create a list, H from the preorder tree walk of this binary heap. The preorder tree walk can be thought of as we are walking along side the minimum spanning tree, always having the vertices to the left of the current direction. Each time we meet a vertex we haven't seen before, we store it in H. Eventually we will return to the starting node.

H is the approximated solution to the TSP problem. An example run can be seen on figure

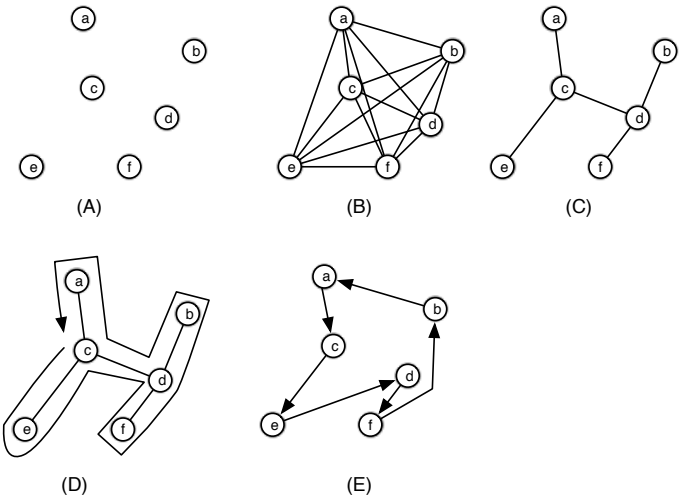

Figure 9 – Example run of the TSP approximation algorithm. (A) initial point cloud. (B) Fully connected graph. (C) Minimum spanning tree. (D) Preorder tree walk. (E) Approximated TSP tour

The approximated TSP tour is a polynomial time 2-approximation algorithm. I.e. the approximation algorithm is maximum twice as 'bad' as the optimal solution.

Proof. The proof is similar to the proof given for the vertex cover problem. We start by showing a lower bound of the optimal tour. Let H^* be a TSP optimal tour. Now remove one edge from H^* we now have a minimum spanning tree. Therefore the total cost of the spanning tree, T, computed in the algorithm above is a lower bound of the optimal TSP tour i.e.

$$c(T) \leq c(H^*) \Leftrightarrow 2c(T) \leq c(H^*) \tag{50}$$

[5]Tree containing all the vertices where the length of the edges is minimized. Running time depends on implementation but can be $\mathcal{O}(E \log(V))$ when using a binary heap

Imagine instead of a preorder tree walk we did a full walk, adding a node to H every time we encountered it disregarding if we have seen the node before. We would then end up with a list of vertices H' where the vertices where allowed to be present multiple times. The full walk traverses every edge in T exactly twice, we have

$$c(H') = 2c(T) \tag{51}$$

i.e. the cost of the full walk is twice the cost of the minimum spanning tree.

We now put the triangle inequality in to play. The triangle inequality states that the shortest path between two nodes is the direct path. I.e. if we remove nodes from H', the cost would decrease.

Now create a new list H from H' where the elements in H is the unique elements in H' in the order they are present in H'. If we compare the cost of H and H' it is clear that

$$c(H) \leq c(H') \tag{52}$$

Combining 50, 51 and 52 we arrive at

$$c(H) \leq c(H') \leq 2c(H^*) \tag{53}$$

which concludes the proof.

□

There exist no approximated solution if the cost function doesn't satisfy the triangle inequality. This follows from the following argumentation: If there where such an approximated polynomial solution with $\rho > 1$ then we could use this approximated solution to solve the Hamilton cycle problem, and since the Hamilton cycle problem is NP-complete this is highly unlikely as we assume the $P \neq NP$. The proof uses reduction to turn the Hamilton cycle problem into the TSP problem, and utilizing that disregarding the size of ρ the algorithm most return a Hamilton cycle.

5.4 Set covering problem

Given a set of elements X and a set of subsets F where each element in F is a subset of X we wish to find the minimum number of subsets in F where the union of these is equal to X. Formally we seek a set of subsets, F^*, where

$$\bigcup_{f \in F^*} f = X \tag{54}$$

and where $|F^*|$ is minimized.

The problem of finding such a set is NP-complete.

Figure 10 shows an instance of the set cover problem.

The greedy-set-cover algorithm is an approximation algorithm. It works as follows

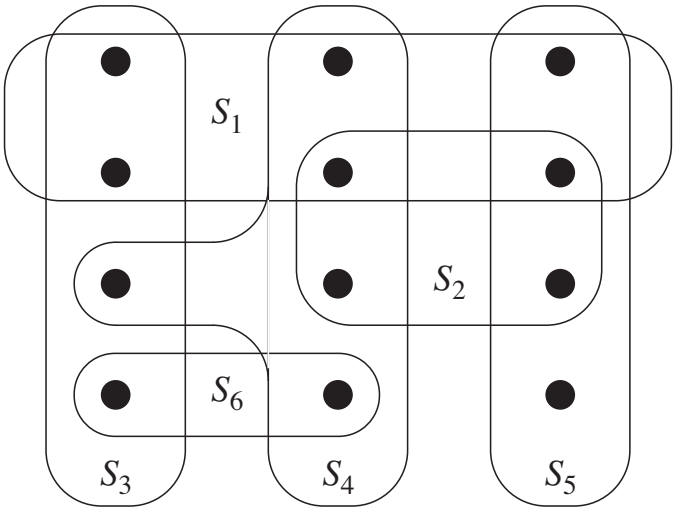

Figure 10 – Instance of the set cover problem, where X consist of the 12 black marks, and $F = \{s_1, s_2, s_3, s_4, s_5, s_6\}$. The greedy algorithm, would start by selecting s_1, then s_4 and s_5 and finally either s_3 or s_6 would be picked in that order. The optimal solution is s_3, s_5, s_6

```
Greedy-set-cover(X,F){
  U = X
  F = ∅
  while(U !=  ∅ ){
    Pick the set S ∈ F such that | S ∩ U | is maximized
    F = F ∪ {S}
    U = U - S
  }
  return F
}
```

The greedy-set-cover algorithm is a polynomial time algorithm $\rho(n)$-approximation algorithm where

$$\rho(n) = H(max()) \tag{55}$$

For the approximation ratio for the greedy SET-COVER algorithms, another simpler proof was given during the lecture. It is based only on the number of remaining uncovered elements which are covered by the next set selected by the greedy algorithm.

Namely, if r is the number of remaining uncovered elements, the next set selected by the greedy algorithm covers at least r/OPT elements, where OPT denotes the number of sets in the optimal solution (i.e. the minimum).

This argument alone suffices to conclude that when the greedy algorithm selects at most OPT sets, at least half of the remaining uncovered elements

are covered. Repeating this argument recursively, yields that all elements are covered after selecting greedily at most $OPT\mathcal{O}(logn)$ sets.

Let n denote the number of elements. In the first iteration we would pick $\geq n/opt$, there would be at least $n - n/opt = n(1 - 1/opt)$ points left. In the second iteration we would pick at least $n(1 - 1/opt)/opt$

http://pages.cs.wisc.edu/~shuchi/courses/787-F07/scribe-notes/lecture02.pdf

Read and understand

5.5 Linear program to vertex cover

Approximate vertex cover an alternative way. Linear programming can be used in approx. algorithms. We want to solve vertex cover in a "weighted" version, called weighted vertex cover. Instead of wanting to archive the minimum number of nodes, we want to archive the minimum weight of the nodes.

Let every node v_n have a weight $w(v_n)$. Goal: To minimize the total weight $\sum_{\forall v \in V} w(v)$ in the vertex cover.

We need to express the problem in terms of integer programming. Thus

$$\sum_{\forall v \in V} w(v)x(v) \tag{56}$$

where $x(v) = 1$ if vertex is in cover otherwise 0. Subject to

$$x(u) + x(v) \geq 1 \qquad \forall (u, v) \in E \tag{57}$$

(At least one of two adjacent vertices has to be in vertex cover.) and

$$x(v) \in \{0, 1\} \qquad \forall v \in V \tag{58}$$

We need to transform the integer programming problem into a linear program by relaxing the last constraint, i.e. $0 \leq x(v) \leq 1 \qquad \forall v \in V$. This linear program can be solved optimal in polynomial time.

The linear program can be related to the integer program as follows: The linear program gives us at least the value of the integer programming problem. The linear program is thus a lower bound of the integer program.

5.6 Subset-sum problem

There are two problems known as the subset sum problem: An optimization and a decision problem. The decision problem is as follows:

Definition 5.1. *Given a set of integers and an integer s, does any non-empty subset sum to s?*

the optimization is as follows

Definition 5.2. *Given a set of integers and an integer s, find a subset of of the integer set so that the sum of the subset is as close to s as possible.*

In both cases the problem is NP-complete.

An exact solution to the subset sum problem is the following algorithm.

```
Exact-subset-sum(S,t){
  n = |S|
  l_0 = {0}
  for(i=1 to n){
    l_i = MergeList(l_{i-1}, l_{i-1} + x_i)  <- add x_i to every element in l_{i-1}
    remove from l_i all elements greater than t.
  }
  return max(l_n)
}
```

where `MergeList(x,y)` as a procedure that merges to list, sorting them in decreasing order an removing duplicates. An example with $S = \{1, 4, 5\}$ yields

```
l_0 = {0}
l_1 = {0,1}
l_2 = {0,1,4,5}
l_3 = {0,1,4,5,6,9,10}
```

The list l_i can grow exponential large, up to 2^i.

The following is a approximation algorithm that returns a value within $1 + \epsilon$ factor of the optimal solution. The idea is to trim the list after each iteration, such that if some elements in the list are within a certain threshold of each other, only the smallest element is retained. This trimming can be done in $\mathcal{O}(m)$ where m is the length of the lust given to the trim function.

```
Approx-subset-sum(S,t){
  n = |S|
  l_0 = {0}
  for(i=1 to n){
    l_i = MergeList(l_{i-1}, l_{i-1} + x_i)  <- add x_i to every element in l_{i-1}
    l_i = Trim(l_i, ε/2n) <- Trim list
    remove from l_i all elements greater than t.
  }
  return max(l_n)
}
```

Similarly, in the exam you will not be required to give an account for the detailed computations in the proof of Theorem 35.8 (SUBSET-SUM), but you should try to understand the idea behind how trimming of the lists is done, and why it works. That is,

(a) on one hand it keeps only a polynomial number of elements in the list, but

(b) at the same time the allowed error is so small, that although it may increase at each step, it still does not add to something which is so big that it violates the required approximation bound.

Understand this

6 Randomized Algorithms

Randomized algorithms, are algorithms that seeks to eliminate the possibility that 'your worst' enemy gives you a input such that the algorithm runs slowly.

6.1 Hiring problem

Say we want to hire a person. We have a external company that gives us the names of the candidates we wish to interview. Each candidate is assigned an integer value describing how good the candidates qualifications are. If the candidate is better qualified i.e. here value is greater than the candidate we presently have, she is hired.

Each time we hire a candidate it cost us money. Interviewing has zero cost.

Now we want to minimize the total cost, when interviewing n candidates. We need to interview all candidates, but every time a new candidate with better qualifications comes around, we need to hire her, and fire the previous. If we have no control over the ordering the candidates arrive in, we risk the situation where the candidates arrive in sorted increasing order by their value. This would mean that we would have to hire all n candidates! The worst case is thus a total cost of $\mathcal{O}(cn)$ where c is the cost of hiring a single candidate.

A randomized approach to this problem would be to receive the full list of candidates from the external company, randomize the list, and call in the candidates in the randomized order. This would eliminate the possibility that the candidates where interviewed in a predetermined manner.

Because the candidates arrive in randomized order, and each candidate is equally likely to be the best, we have that the probability that the $i'th$ candidate is hired can be expressed as $1/i$.

From the harmonic series we have the cost as $1/1+1/2+1/3+\ldots+1/n \approx ln(n)$ which is also the *expected* cost of the algorithm.

In order to arrive at this result we used the *linearity of expectations*. Linearity of expectations means that the expected value of a sum equals the expected value of each i'th terms of the sum. Dice example.Throwing a dice yields a number in the interval $1-6$. The expected number is $3,5$. Throw the dice 10 times then the expected sum will be 35. This holds only for summation not products. For products it only holds for unrelated events (iid).

6.2 Randomization

Many algorithms take as input an array of data. This data can be randomized in several ways. Two methods are described here. In both cases it is worth noticing that the probability of the random generator is uniform!

6.2.1 Permute by sorting

Permute by sorting works in the following way. Generate a new array, P, with length n where n is the length of the input array A. Each entry in P has a

random value in the range $[1, n^3]$. Now sort A by the keys of P. The complexity of this algorithm is as follows. Generating P takes $\mathcal{O}(n)$. Sorting can be done in $\mathcal{O}(n\log(n))$. The total complexity is thus $\mathcal{O}(n + n\log(n)) = \mathcal{O}(n\log(n))$.

An example of the algorithm is as follows. Given $A = \langle 1, 2, 3, 4\rangle$, we compute $P = \langle 36, 3, 62, 19\rangle$. By sorting A with the keys in P we arrive at $A = \langle 2, 4, 1, 3\rangle$.

6.2.2 Randomize in place

Randomize in place works as follows. Given an array A, we swap the $i'th$ element of A with a element picked randomly from the interval $[i, n]$ where n is the number of elements in A. This randomization algorithm runs in $\mathcal{O}(n)$.

6.3 Quicksort

Quicksort has a worst case running time of $\mathcal{O}(n^2)$. This running time occurs, when the pivot is chosen as the largest element at every iteration. In order to accommodate this problem we can randomize the way we choose the pivot.

The changes to quicksort are small. Instead of always choosing the element in the last position of the array we are currently working on, then choose the pivot at random. We thus swap the last element with a random element from the array and choses this. The only change to the quicksort algorithm is thus the swap procedure. The expected running time is now $\mathcal{O}(n\log(n))$.

6.4 Selection algorithm

Randomized select is a randomization algorithm that finds the $i'th$ smallest element in time $\mathcal{O}(n)$. It relies on the randomized partition function also used in quicksort.

The algorithm is recursive and selects a part of the total array where it knows the $i'th$ smallest element is in. Following is the algorithm

```
Randomized Select(A, p, r, i){
  if p==r                                        <- Base case
      return A[p]
  q = randomized_partition(A,p,r)                <- Partition A, q is index into the
  k = q-p+1                                      <- k is the
  if k==i
      return A[q]                                <- BINGO
  elseif i<k
      return Randomized Select(A, p, q-1, i)     <- Look in the left part
  else
      return Randomized Select(A, q+1, r, i-k)   <- Look in the right part
}
```

The algorithm starts by checking if the list only consist of 1 element. Otherwise it does a randomized partition of the list and returning the index of the partitioning in q. k can be interpreted as the number of elements between p and q. If $k = i$

then we know that the we have found the element, and the element is the same as the pivot element.

If $i < k$ then we know that the $i'th$ element is to the left of the pivot element. We recursively call the function of this part of the list, discarding the current pivot element.

If $i > k$ then we know that the $i'th$ element must lie behind the pivot element, and we thus look at this part of the list. When looking at the right hand side of the list, we need to change i so it becomes $i = i - k$.

Figure 11 shows an example run of the algorithm. The expected running time of the algorithm is $\mathcal{O}(n)$.

6.5 Binary search tree

A binary search tree is a tree where each node has either none, one or two children. A child positioned to the left of a node has a value less than the node, and a child positioned to the right of a node has a value greater than the node.

The height of the binary search has direct impact of the time it takes to search, insert and delete nodes. Therefore it is desirable that the height is as small as possible, i.e. the binary tree must be balanced.

A binary search tree can be built from an array of values. The first element of the array becomes the root of the tree. The second element becomes a child of the root, and is positioned to the left if it is lesser than the root node, otherwise it is positioned to the right. This procedure poses a problem. If your worst enemy gave you a an array sorted in increasing[6] order the binary search tree would have height n, where n is the number of elements in the array. This is not tractable!

The reason is that whenever a new node is inserted to the search tree it is positioned to the right, because the element being inserted is greater than all other elements inserted. An illustration can be seen on figure 12

We can accommodate this problem, by either choosing the element to insert randomly from the array, or totally randomize the array before inserting.

[6] or decreasing, the height is still $\mathcal{O}(n)$

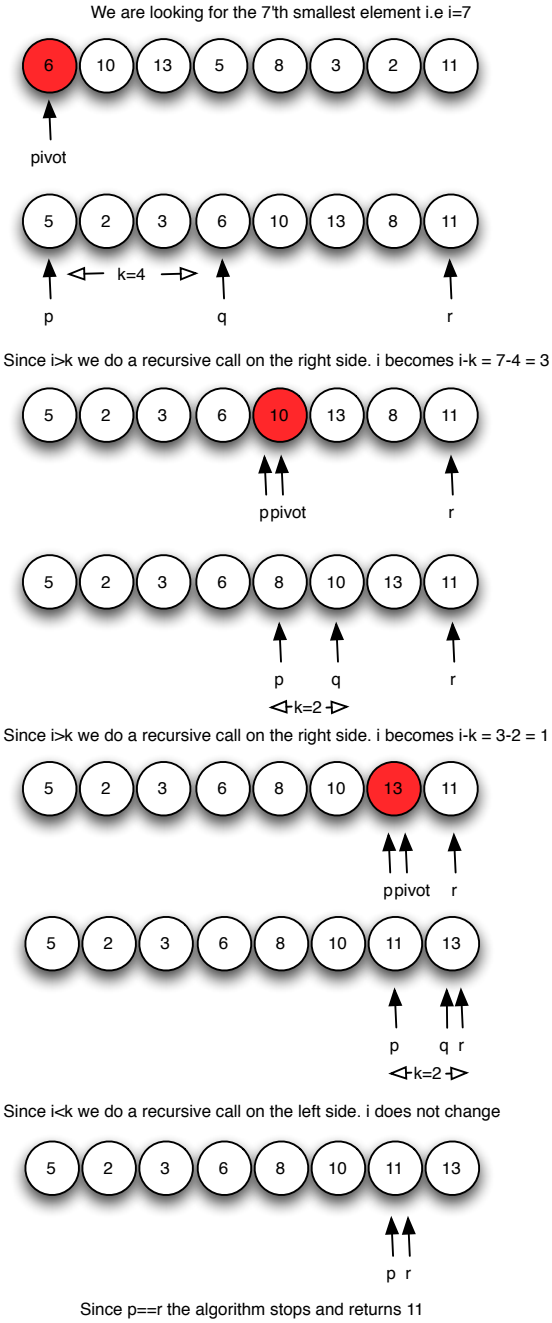

Figure 11 – Example run of the randomized select algorithm

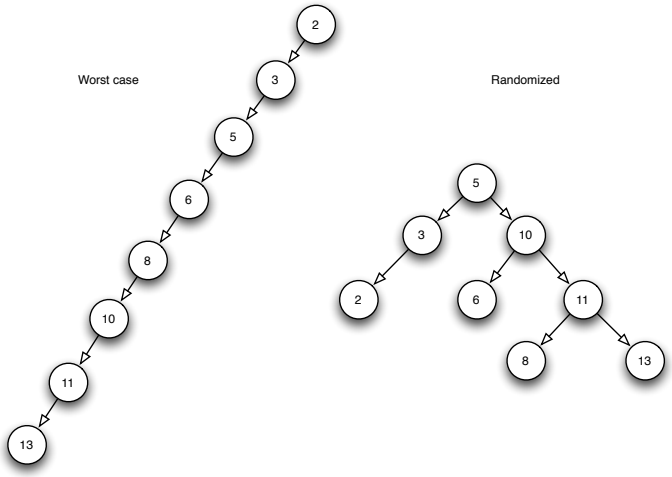

Figure 12 – Illustration of the difference in height, when using the randomized version, compared to the original version.

7 Computational Geometry, Convex hulls

Computational Geometry, Convex hulls, CLRS, Chapter 33.3 and the papers by [Kirkpatrick&Seidel] and [Chan] (only 2D case in both articles).

Given a set of points P, the convex hull problem seeks to find a convex set containing all the points in P.

Let S be a convex set. Then for any two points $a, b \in S$ the following is true

$$(t-1)a + tb \qquad t \in [0,1] \tag{59}$$

thus, there exists a line between any two points in S, such that the line between them is contained within S.

In the following we assume that no points has the same x- or y- coordinates and no three points are collinear. This is true in the continous world, but watch out when going into the discrete world of computers!

We denote the convex hull of a set S to be $CH(S)$. There exists several algorithms to calculate the convex hull. Int he following, Grahams scan, Jarvis' march, quickhull, randomized incremental and marriage before conquest is explained in detail.

7.1 Grahams scan

Grahams scan has a running time of $\mathcal{O}(n \log(n))$. It starts by choosing a the point $p_0 \in S$, with the lowest y-coordinate. This can be done in $\mathcal{O}(n)$. It then sorts the remaining points $p_i \in S$ by the angle between the line (p_0, p_i) and the x-axis. This sorting can be done in $n \log(n)$ by e.g. heap sort[7]

Put p_0, p_1, p_2 in the list of nodes that constitutes the convex hull. If making a left turn when adding p_3, remove p_2 and add p_3 to the list. Otherwise if making a right turn, add p_3 to the list. Continue adding nodes. In general, if adding node p_j creates a left turn, then remove node p_{j-1} and add p_j. Otherwise just add p_j. When last node has been reached, the convex hull is done.

The correctness of Grahams scan can be explained by the following observations.

- Grahams scan will never go backwards behind the initial node.
- When arriving at point p_i all points between the initial node and the point p_i are right turns on the polygonal line constructed so far
- After arriving at the initial node by a right turn, we get a polygonal line consisting of purely right turns.

Figure 13 shows an example run of Grahams scan.

The time complexity is $\mathcal{O}(n \log(n))$ since sorting takes $\mathcal{O}(n \log(n))$ and the scanning takes $\mathcal{O}(n)$, since each point is only considered once. The total time complexity is thus $\mathcal{O}(n \log(n) + n) = \mathcal{O}(n \log(n))$.

[7] Heap sort uses a heap, i.e. a tree datastructure where the largest element is on top.

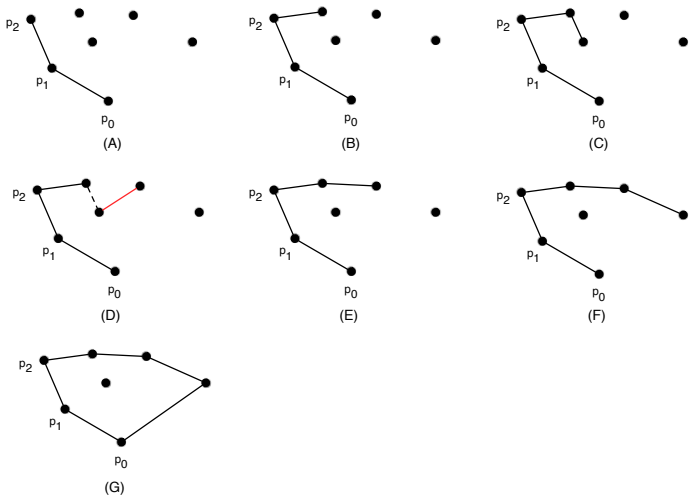

Figure 13 – Example run of Grahams scan

7.2 Jarvis' march

Jarvis' march is also knows as the 'Giftwrapping algorithm'. It has a complexity of $\mathcal{O}(nh)$, where n is the number of nodes, and h is the number of nodes in the convex hull. Since part of the complexity relies on the final output, the algorithm is *output sensitive* and can be $\mathcal{O}(n^2)$ when $h = n$.

Find the point, p_0, which is placed leftmost. This point can be found in $\mathcal{O}(n)$. Add p_o to the convex hull list. Now find the point p_1 which has every other point p_i to the right and add it to the list of convex hull nodes. This can also be done in $\mathcal{O}(n)$ by comparing polar angles from p_0. Continue to add p_i to the list such that every other node is to the right of p_i. Continue until the $p_i = p_0$.

The complexity of Jarvis' march can be computed as the time it takes to find p_i times the number of nodes, h in the convex hull. The complexity is thus $\mathcal{O}(nh)$.

An example run of Jarvis' march can be seen on figure 14

7.3 Quickhull

Quickhull is an algorithm that has an expected running time of $\mathcal{O}(n \log(n))$, and a worst case running time of $\mathcal{O}(n^2)$. The algorithm starts by finding the leftmost and rightmost points, A and B. It then draws a line between these two points, and splits the point set S into two. Let the set S_1 contain the points above the line, and S_2 contain the points below the line.

For each of these two sets we proceed recursively.

Find the point, P farthest away from the line. the convex hull must contain P, so insert the point between A and B. A triangle is formed by ABP. Remove all points from S that is contained within this triangle. The cross product can

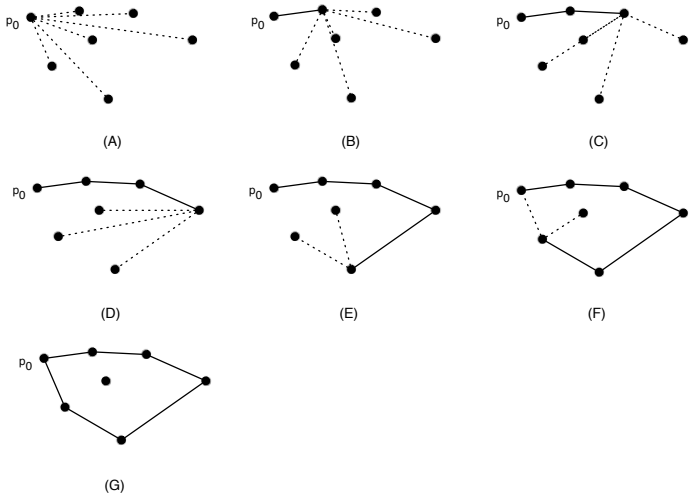

Figure 14 – Example run of Jarvis' march. The dotted lines are the polar angles

be used to calculate wether a point lies inside a triangle.

We form two new sets. One set containing the set of points above the line AP. The other set containing points above the line BP. The line AB is then replaced by these two lines. and the algorithm proceeds recursively. The algorithm stops when the sets are empty or if it only consist of one node.

An example run of Quickhull can be seen on figure 15

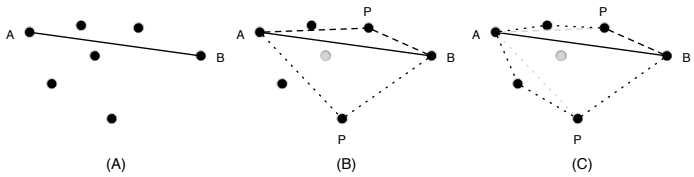

Figure 15 – Example run of quickhull. (A) find the left and right most points. (B) Recursively process the two subsets and find the point farthest away form the line. (C) Repeat recursively

If the partitioning yields balanced sets, the the expected running time is $\mathcal{O}(n \log(n))$. If the partitioning is extremely unbalanced then the running time is $\mathcal{O}(n^2)$. A example of an extremely unbalanced set of points, is when the points lie on a half circle.

7.4 Marriage before conquest (MBC)

The Kirkpatrick-Seidel algorithm (Marriage before conquest) is output sensitive and has running time $\mathcal{O}(n \log h)$, where h is the number of points in the convex hull.

The algorithm first calculates the upper hull and then the lower hull. It then merges these two into the final convex hull. We can safely assume that the merging of the two half hulls can be performed in $\mathcal{O}(1)$.

- Let S be the set of points. Divide S into two sets, by dividing S with the line (p_j, p_k), where $p_j \in S$ is the point with minimum x-value and $p_k \in S$ is the point with maximum x-value. Let P be one of the two sets.
- The algorithm then calculates the median x-coordinate M of the points in P. It the finds the bridge segment that crosses the vertical line $x = M$. It finds this bridge segment by a technique called Prune & Search. $x = M$ divides P in half. The bridge segment across this line, will be part of the final convex hull.
- Delete the points under the bridge, and split P into the two half's divided by the vertical line.
- Continue recursively on each half.

The Prune & Search method works in the following way

- Randomly pair all the points into line segments.
- Determine the median slope, m, of all the distinct line segments. If the number of points is odd, there will be a point that is not part of a line segment. Give this point a slope of 0. If the number of line segments is even there are two choices for the median slope. Let m be the max of these two slopes.
- Construct a sweep line, L, having the median slope, i.e $y = mx + b$. Find a point p_t such that L is a supporting line[8] for P at p_t, this can be done by translating. We call p_t the top point.
- Let $p_j.x$ define the x-coordinate of the point p_j.
  - If $p_t.x \geq M$, i.e the top point is to the right of the vertical line M, then for each line segment with slope $< m$ remove the right point of the line segment, i.e q_{ir}.
  - If $p_t.x < M$, i.e. the top point is to the left of the vertical line M, then for each line segment with slope $> m$, remove the left point q_{il}.
- Repeat this until only two points are left. We know that at least half of the line segments has a slope greater than m, therefore we can conclude that we at each iteration of the bridge finding removes $1/4$ of the points. Note that the points are only removed in the current step where we find the bridge, not the set P.

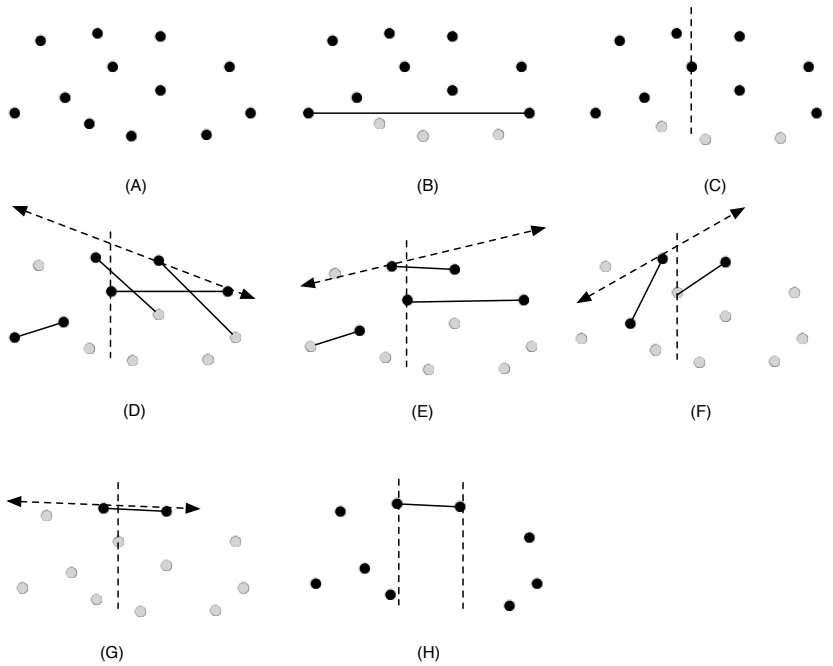

Figure 16 – (A) Original pointset. (B) Separating the set in two. (C) Finding the median. (D) Pairing points, finding support line and removing points (grey color points are deleted). (E) Removing points. (F) Removing points. (G) Bridge found. (H) Deleting points under bridge

Figure 16 shows how to fine the bridge

The complexity of the bridge finding is as follows. Pairwise point matching takes $\mathcal{O}(n)$, finding the median slope takes $\mathcal{O}(n)$, slope translation takes $\mathcal{O}(n)$. Each time we remove points we are sure to remove $1/4$ of the points. This yields

$$\begin{aligned} T(n) &= \mathcal{O}(1) && n = 2 \\ T(n) &= T(3n/4) + \mathcal{O}(n) && \text{if } n > 2 \end{aligned}$$

thus the complexity of the bridge finding and finding the median is $T(n) = \mathcal{O}(n)$.

The total overall complexity can be bounded by the following function

$$\begin{aligned} f(n,h) &= cn && h = 2 \\ f(n,h) &= cn + \max_{h_l+h_r=h} \{f(n/2,h_l) + f(n/2,h_r)\} && h > 2 \end{aligned}$$

where c is a positive constant. The max is introduced because the algorithm is output sensitive. The claim is that the complexity is $f(n,h) = \mathcal{O}(n\log(h))$, thus we can find a upper bound by the function $cn\log(h)$.

[8] A supporting line for a set S is a line that contains a point $p_i \in S$ an no other points above.

Proof. For $h = 2$

$$f(n,h) = c_1 n \leq cn\log(2) \tag{60}$$

this trivially holds if $c_1 \leq c$

Assume $h > 2$ and bu subsitution we have

$$\begin{aligned} f(n,h) &= cn + \max_{h_l+h_r=h}\{cn/2\log(h_l) + cn/2\log(h_r)\} \\ &= cn + cn/2 \max_{h_l+h_r=h}\{\log(h_l) + \log(h_r)\} \\ &= cn + cn/2 \max_{h_l+h_r=h}\{\log(h_l h_r)\} \\ &= cn + cn/2 \max_{h_l+h_r=h}\{\log(h_l(h-h_l))\} \\ &= cn + cn/2 \max_{h_l+h_r=h}\left\{\log(hh_l - h_l^2))\right\} \end{aligned}$$

When does the function $g(h_l) = \log(hh_l - h_l^2)$ takes it's maximum. It does when the derivative is 0.

$$g'_{h_l}(h_l, h) = \frac{1}{hh_l - h_l^2}(h - 2h_l) = 0 \tag{61}$$

If $g'_{h_l}(h_l) = 0$ then $h - 2h_l = 0$ yielding $h_l = \frac{h}{2}$. Thus

$$\begin{aligned} f(n,h) &= cn + cn/2\log(h\frac{h}{2} - \frac{h^2}{2^2})) \\ &= cn + cn/2\log(\frac{h^2}{2} - \frac{h^2}{2^2})) \\ &= cn + cn/2\log(\frac{2h^2}{4} - \frac{h^2}{2^2})) \\ &= cn + cn/2\log(\frac{h^2}{2^2})) \\ &= cn + 2cn/2\log(\frac{h}{2}) \\ &= cn + cn\log(\frac{h}{2}) \end{aligned}$$

which has a running time of $\mathcal{O}(n\log(h))$

□

7.5 Chan and relations to MBC

Chan is an algorithm that combines Grahams scan and Jarvis' march (giftwrapping). It is interesting because it involves combining two slower algorithms together to form an algorithm that is faster than either one.

The problem with Graham's scan is that it sorts all the points, and hence is doomed to having an Omega(n log n) running time, irrespective of the size of

the hull. On the other hand, Jarvis's march can perform better if you have few vertices on the hull, but it takes $\mathcal{O}(n)$ time for each vertex in the hull.

The algorithm works as follows:

First suppose we know there are h points on the convex hull, this algorithm starts by shattering the input points in n/h arbitray subsets, each of size h, and computing the convex hull of each subset using Graham's scan. This much of the algorithm requires $\mathcal{O}((n/h)h\log(h)) = O(n\log(h))$ time.

Once we have the n/h subhulls, we follow the general outline of Javis's march, wrapping a string around the n/h subhulls. Start with the leftmost input point l, starting with $p = l$ we successively find the convex hull vertexes in counter-clockwise order until we return back to the original leftmost point again.

The successor of p must lie on a right tangent line between p and one of the subhulls, a line from p through a vertex of the subhull, such that the subhull lies completely on the right side of the line from p's point of view. We can find the right tangent line between p and any subhull in $O(\log(h))$ time using a variant of binary search. Since there are n/h subhulls, finding the successor of p takes $\mathcal{O}((n/h)\log(h))$ time together. Since there are h convex hull edges, and we find each edge in $\mathcal{O}((n/h)\log(h))$ time, the overall running time of the algorithm is $\mathcal{O}(n\log(h))$.

Unfortunately, this algorithm only takes $O(n\log(h))$ time if we know the value of h in advance. So how do we know h's value? Chan's trick is to guess the correct value of h, let's denote the guess by h'. Then we shatter the points into n/h' subsets of size h', compute their subhulls, and then find the first h' edges of the global hull. If $h < h'$, this algorithm computes the complete convex hull in $O(n\log(h'))$ time. Otherwise, the hull doesn't wrap all the way back around to l, so we know our guess h' is too small.

Chan's algorithm starts with the optimistic guess $h' = 3$. If we finish an iteration of the algorithm and find that h' is too small, we square h' and try again. In the final iteration, $h' < h^2$, so the last iteration takes $\mathcal{O}(n\log(h')) = \mathcal{O}(n\log(h^2)) = \mathcal{O}(n\log(h))$ time.

The total running time of Chan's algorithm is given by the sum: $\mathcal{O}(n\log(3) + n\log(3^2) + n\log(3^4) + ... + n\log(3^{2^k}))$, for some integer k. We can rewrite this as a geometric series: $\mathcal{O}(n\log(3) + 2n\log(3) + 4n\log(3) + ... + 2kn\log(3))$. So Chan's algorithm runs in $\mathcal{O}(n\log(h))$ time overall, even when we don't know the value of h.

Example run of chan

8 Computational Geometry, Delaunay triangulatiom

8.1 Applications

Triangulation is used in graphics and movies. It can also be used to model terrain when the terrain is represented as a bunch of sample points where each sample point is representing the height of the terrain compared to, lets say, sea level. A terrain visualized solely by height samples is not that interesting an doesn't look very natural! Instead we can do triangulation of the samples in 2D, and after the triangulation lift the points op in 3D yielding a triangulated surface, representing a terrain.

The central question here is how we triangulate the points? and if there are multiple methods how do we choose the method that gives us the most appropriate triangle net? In the following subsections we will answer these two questions.

8.2 Definition

What do we mean by a good triangulation? Simple case is to connect one point to every other point. This is NOT a good triangulation. The ideal triangulation is a triabgulation that is ballanced, i.e. the we want to avoid small angles.

We define angle optimality as follows.

Let T be a triangulation of P. Let T consist of m triangles. Then T has $3m$ angles. Let the angles be in a sorted vector in increasing order. Denote this vector $A(T)$ We call this vector an *angle vector*. Let T' be another triangulation of P. We say that T is *angel-optimal* if T is lexicographically larger than $\forall T' \in P$.

Lexicographically larger means that for some index i the angles after i in $A(T)$ is larger than the angels in $A(T')$. E.g.

$$\begin{pmatrix} 1 \\ 3 \\ 5 \\ 7 \end{pmatrix} > \begin{pmatrix} 1 \\ 2 \\ 5 \\ 7 \end{pmatrix} \tag{62}$$

the reason for this definition, is we want as uniform angles as possible in all triangles.

We define maximal planar subdivision as

Definition 8.1. *Maximal planar subdivision: A subdivision S such that no edge connecting two vertices can be added to S without destroying it's planarity, i.e. any edge not in S intersects and edge $e \in S$.*

from the maximal planar subdivision we can define the triangulation as

Definition 8.2. *Triangulation of set of points P: A maximal planar subdivision whose vertices are elements of P.*

8.3 Edge flip

Given two adjacent triangles $p_1p_2p_4$ and $p_2p_3p_4$, one can do an edge flip as follows. Remove the common edge by creating two new triangles consisting of $p_1p_3p_4$ and $p_1p_2p_3$. An illustration of an edge flip can be seen on figure 17.

In the two adjacent triangles there are 6 angles. If we can increase the minimum angle by doing a edge flip on the shared edge, we called the shared edge illegal.

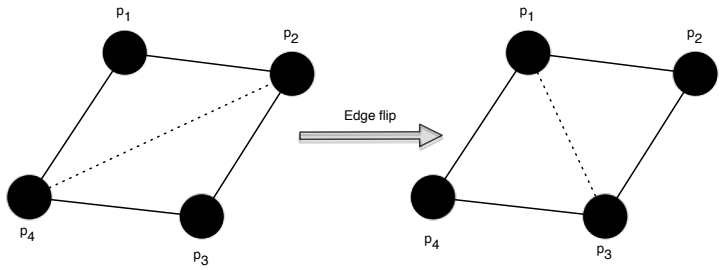

Figure 17 – Example of an edge flip

Instead of checking wether an edge is illegal by computing all the angles, we can use the following observation

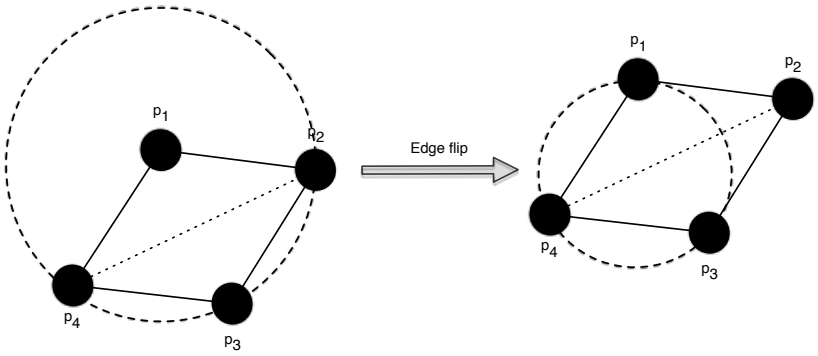

Figure 18 – If p_1 is inside the circumcircle then the edge between the two triangles is illegal an we perform an edge flip.

8.4 Delaunay triangulation

The Delaunay triangulation for a set of points P is a triangulation where no point $p_i \in P$ is inside the circumcircle of any triangle in the triangulation set. The Delaunay triangulation is actually the dual of the Voronoi diagram. figure 19 shows this relation.

The Voronoi diagram is shown as red on figure 19, while the triangulation is shown as black. A line in the Voronoi diagram separating two vertices, indicates

that there is an edge between the two vertices in the triangulation.

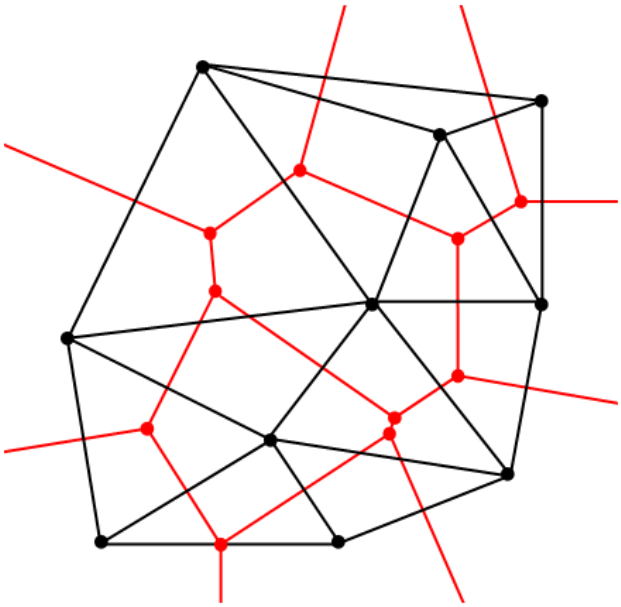

Figure 19 – Duality between the Delaunay triangulation and the Voronoi diagram.

Returning to the legal and illegal edges. Let P be a set of points in the plane then a triangulation of P is legal, *if and only if*, the triangulation is a Delaunay triangulation.

8.5 Naive method

Given a set of points, *take the convex hull*, add diagonals (edges that connects vertices), without crossing those we already have drawn, until there are no more vertices. The result is a triangulation. I.e. we partition the points into triangles.

In order to make the triangulation into a Delaunay triangulation, we can do edge flips until there are no more illegal edges.

The most straightforward way of efficiently computing the Delaunay triangulation is to repeatedly add one vertex at a time, retriangulating the affected parts of the graph. When a vertex v is added, we split in three the triangle that contains v, then we apply the flip algorithm. Done naively, this will take O(n) time: we search through all the triangles to find the one that contains v, then we potentially flip away every triangle. Then the overall runtime is O(n2).

8.6 Computing the Delaunay triangulation

The algorithm is randomized incremental algorithm, adding one point at a time.

Let P be the set of points we want to triangulate. Find the point with the highest y coordinate, and name this point p_0. Make a big triangle, that contains all the points in P such that p_0 is one of the corners in the triangle.

Now pick a random point from P and add it to the big triangle, such that the big triangle is divided into 3 sub triangles. Run a legalize edge procedure, ensuring that all the edges are legalized.

Keep adding points from the set P, and legalize the edges of until P is empty.

As a final stage remove the points p_{-1} and p_{-2} along with the incident edges.

THe data structure used is a point location structure. It is a tree like structure where the nodes defines the position of previous triangles and the leafs corresponding to the current visible triangles.

Given a point the data structure makes it possible to locate the triangle of which the point is placed upon.

Figure 20 shows how the data structure is maintained when splitting existing triangles and flipping edges.

The *expected* running time of the algorithm is $\mathcal{O}(n \log(n))$. This can be explained as, it takes constant time to create new triangles the maximal number of triangles created is $\mathcal{O}(n)$. When identifying the triangle in which a point is located, we use the point location data structure, and can make this identification in $\mathcal{O}(\log(n))$.

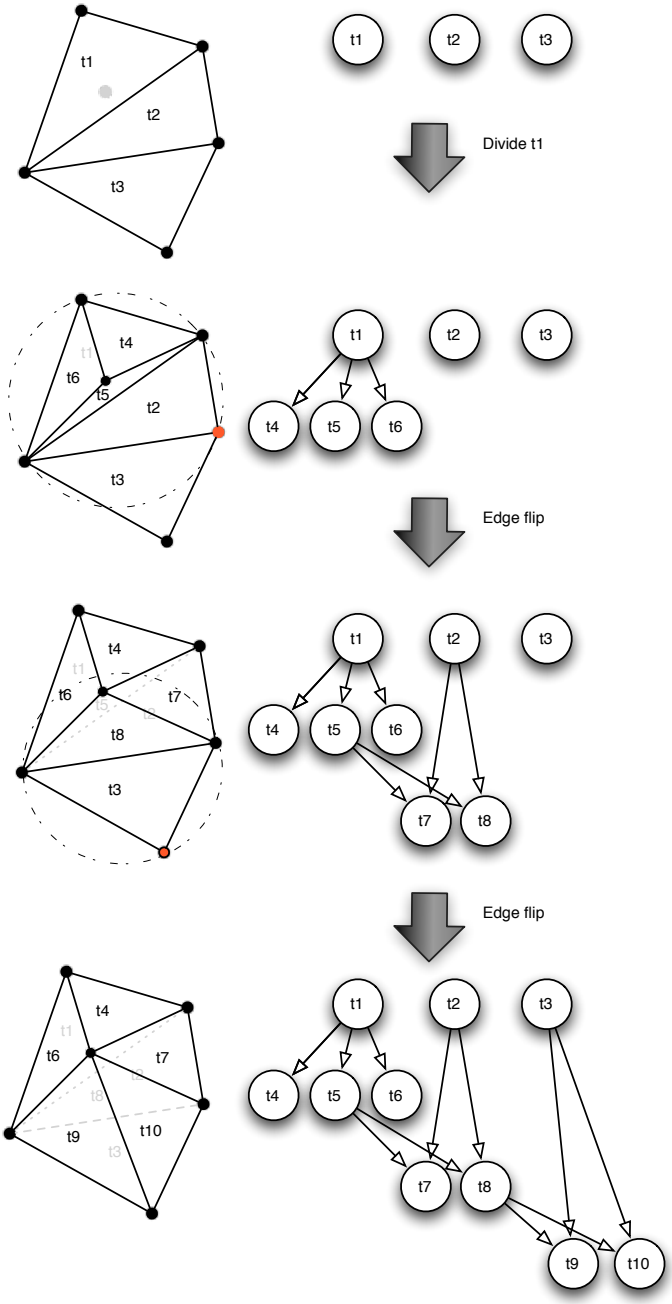

Figure 20 – Example of maintaining a data structure when flipping edges and dividing triangles.