Examn notes for Advanced Algorithms and Datastructures 2014

Martin Jørgensen

August 14, 2014

Contents

Dispositions

Max-Flow

1. (Define a Flow Network)

    - (Capacity Constraint)
    - (Flow Conservation)

2. Define a Max Flow

3. (How to have multible source/sink networks)

4. (Introduce Residual Networks)

5. (Introduce Augmenting Paths)

6. Cuts - In particular the min cut max flow

7. Introduce Ford-Fulkerson / Edmonds-Karp

Fibonacci Heaps

1. Mergeable heaps
2. Structure
3. Operations
    - Make-Heap
    - Insert
    - ExtractMin
    - Union/Merge
    - DecreaseKey
    - Delete

Wrwite something about $D(n)$ since it's used for all the proofs.

NP-Completeness

1. Polynomial time vs Superpolynomial time
2. P-Class problems
3. NP-Class problems
4. Decisions vs Optimization problems
5. Reductions and verifiability/certificates
6. P vs NP vs NPC

Randomized Algorithms

1. Las-Vegas vs. Monte Carlo Algorithms
2. Decision Monte Carlo, one sided vs. two sided error.
3. Bounding Runningtimes
4. Markov & Chebyshev's inequalities
5. Randomized Quicksort
6. (Randomized Selection)

Hashing

1. Universal Hashing
2. Simple Hash Tables w. Chaining
3. Signature Hashes
4. Multiply-mod-prime
5. Strong Universality

Exact Exponential Algorithms & Fixed-parameter tractable problems

1. O^*-notation
2. Parameterized Complexity
3. Travelling Salesman Problem using Dynamic Programming
4. CNF-Satisfiablility.
5. I DON'T KNOW WHAT TO DO!?

Approximation Algorithms

1. Performance Ratios
2. Approximation Schemes
3. Vertex Cover example
4. Traveling Salesman example

Computational Geometry

1. Terrains

2. What is Triangulation

3. Illegal Edges

4. Delaunay Triangulation

Linear Programming and Optimization

1. WHOOP

Notes

Max-Flow

Flow Network

A flow network $G = (V, E)$ is a directed graph where each edge $(u, v) \in E$ have a nonnegative capacity $c(u, v) \geq 0$. In addition, for any edge (u, v) there can be no antiparallel edge (v, u).

Two vertices in the network have special characteristics the source s and sink t. We assume each vertex $v \in V$ lies on some path from s to t, that is, for each vertex $v \in V$, the flow network contains a path $s \leadsto v \leadsto t$.

Flow Definition

We have a flow network $G = (V, E)$ with a source s and a sink t, the network has a capacity function $c(u, v)$. A flow is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies the two following properties:

- **Capacity Constraint**:
  For all $u, v \in V$, we require $0 \leq f(u, v) \leq c(u, v)$

- **Flow Conservation**:
  For all $u \in V - \{s, t\}$ we require

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

  When $(u, v) \notin E$, there can be no flow from u to v, and $f(u, v) = 0$. We call the nonnegative quantity $f(u, v)$ the flow from vertex u to vertex v.

The value $|f|$ of a flow f is defined as

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

that is the total flow out of the source minus the flow into the source. For an example of a flow see Figure 1

Antiparallel Edges and Multiple Sources/Sinks

Since a flow network cannot contain anti-parallel edges, but we want to be able to represent them in our graph, we need a way to do so. This is done by inserting an additional node v' and let one of the edges go through this node instead, see Figure 2 for an example.

If a network have multiple sources or sinks, we can convert it to a single source/sink network by adding a supersource and supersink. An example of such conversion can be seen in Figure 3.

Flow Examples

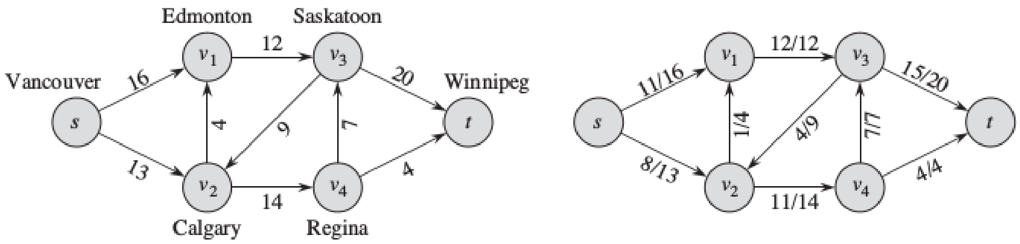

Figure 1: Example flow.

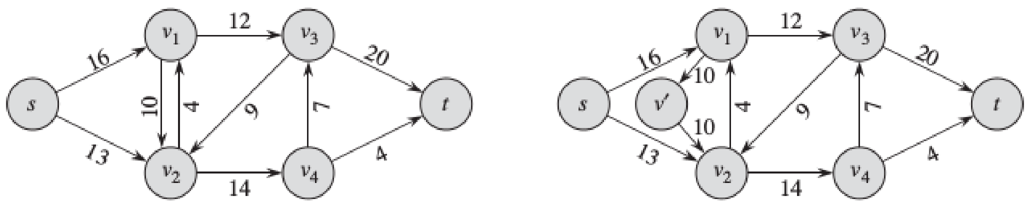

Figure 2: Conversion from antiparallel edges to proper flow.

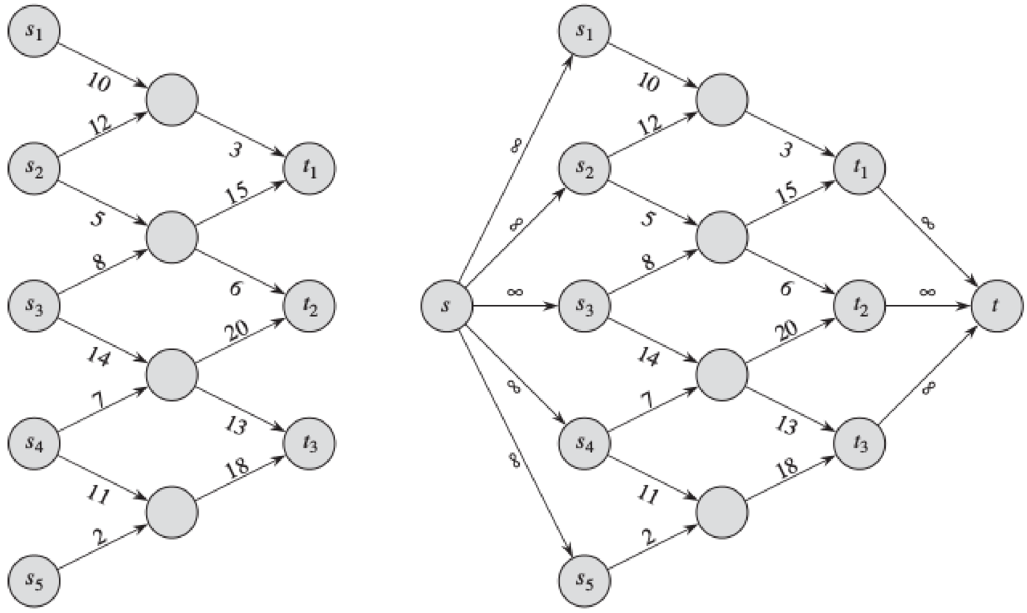

Figure 3: Example of a graph with multiple sources and sink, combined using a supersource and supersink.

Residual Networks

Given a flow network G and a flow f the residual network G_f consists of edges and capacities that represent how we can change the flow on edges of G. Suppose we have a flow network $G = (V, E)$ with source s and sink t. Let f be a flow in G, and consider a pair of vertices $u, v \in V$. We then define the residual capacity $c_f(u, v)$ like this:

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

Given a flow network $G = (V, E)$ and a flow f, the residual network og G induced by f is $G_f(V, E_f)$, where

$$E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$$

An example of a residual network can be seen in Figure 4.

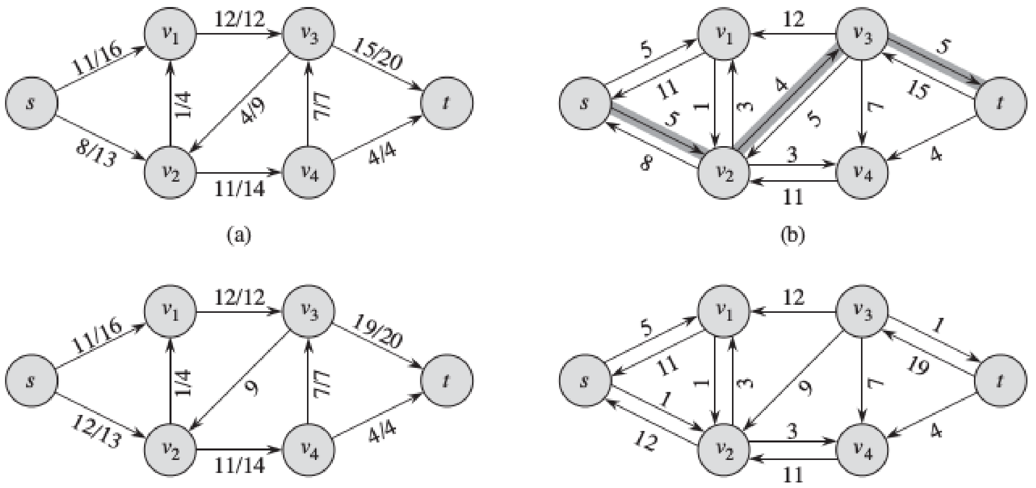

Figure 4: An example of a flow being augmented and showing the residual graph.

Augmenting Flows

Augmenting paths are simply flows that can be added to other flows in order to increase the flow value through the network. Augmenting flows are described using the $\uparrow$ operator like so:

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

an example of an applied augmenting path can bee seen in Figure 4.

Lemma 26.1 Let $G = (V, E)$ be a flow network with source s and sink t, and let f be a flow in G. Let G_f be the residual network of G be induced by f, and let f' be a flow in G_f. Then the function $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f'|$.

Augmenting Paths

Given a network $G = (V, E)$ and a flow f, an augmenting path is a simple path from s to t in the residual network G_f. The shaded path in Figure 4(b) is an augmenting path. We can increase the flow on each edge in the augmenting path p by an amount equal to the residual capacity of p given by

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

That is, the smallest amount of spare capacity on any edge in p.

Lemma 26.2 let $G = (V, E)$ be a flownetwork, let f be a flow in G, and let p be an augmenting path in G_f. Define a function $f_p : V \times V \to \mathbb{R}$ by

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

Then, f_p is flow in G_f with value $|f_p| = c_f(p) > 0$.

Corollary 26.3 let $G = (V, E)$ be a flownetwork, let f be a flow in G, and let p be an augmenting path in G_f. Let f_p be defined as in Lemma 26.2, and suppose that we augment f by f_p. Then the function $f \uparrow f'$ is a flow in G with value $|f \uparrow f'| = |f| + |f_p| > |f|$.

Cuts in flow networks

A cut (S, T) of a flow network $G = (V, E)$ is a partition of V into S and $T = V - S$ such that $s \in S$ and $t \in T$. If f isa a flow, then the net flow $f(S, T)$ across the cut (S, T) is defined to be

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

The capacity of the cut (S, T) is

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

A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network. Note that for capacity we count only edges going from S to T while for flow we count edges going both directions. Lemma 26.4 shows that for a given flow f, the net flow across any cut is the same and it equals $|f|$, the value of the flow.

Lemma 26.4 Let f be a flow in a flow network $G = (V, E)$ with source s and sink t, and let (S, T) be any cut of G. Then the net flow across (S, T) is $f(S, T) = |f|$

Corollary 26.5 The value of any flow f in a flow network G is bounded from above by the capacity of any cut of G.

Max-flow min-cut theorem:

Theorem 26.6 If f is a flow in a flow network $G = (V, E)$ with source s and sink t, then the following conditions are equivalent:

1. f is a mximum 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.

Proof (1) $\implies$ (2): If we assume f is a maximum flow but there still is an augmenting path p in the residual graph G_f, then by Corollary 26.3, the flow found by $f \uparrow f_p$ is a flow with value strictly greater than |f|, contradicting the assumption that f is a max flow.

(2) $\implies$ (3): Suppose G_f has no augmenting paths, that is, that G_f contains no paths from s to t. We 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 a cut: we have $s \in S$ trivially, and $t \notin S$ because there is no path from s to t in G_f.

We now consider a pair of vertices $u \in S$ and $v \in T$. If $(u, v) \in E$, we must have $f(u, v) = c(u, v)$ otherwise $(u, v) \in E_f$ wich would place v in S.

If $(v, u) \in E$, we must have $f(v, u) = 0$, because otherwise $c_f(u, v) = f(v, u)$ would be positive and we would have $(u, v) \in E_f$, whouch would place v in S. If neither (u, v) or (v, u) is in E, then $f(u, v) = f(v, u) = 0$. We thus have

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

By Lemma 26.4, we then have $|f| = f(S,T) = c(S,T)$.

(3) $\implies$ (1): By corollary 26.5, $|f| \le c(S,T)$ for all cuts (S,T). The condition $|f| = c(S,T)$ thus implies that f is a maximum flow. □

Ford-Fulkerson

The general algorithm:

FORD-FULKERSON-METHOD(G,s,t)

1 initialize flow f to 0
2 **while** there exists an augmenting path p in the residual network G_f
3   augment flow f along p
4 **return** f

Implementation:

FORD-FULKERSON(G,s,t)

1 **for** each edge $(u,v) \in G.E$
2     $(u,v).f = 0$
3 **while** there exists a path p from s to t in the residual network G_f
4     $c_f(p) = \min\{c_f(u,v) : (u,v) \text{ is in } p\}$
5   **for** each edge (u,v) in p
6   **if** $(u,v) \in E$
7             $(u,v).f = (u,v).f + c_f(p)$
8   **else** $(v,u).f = (v,u).f - c_f(p)$

Assume we can pick the path p in linear time, the loop header is running in $O(E)$. If f^* denote a maximum flow, then the while loop is executed at most $|f^*|$ times, since each augmentation must increase the flow value with at least one. The for loop inside the loop can be done in $O(E)$ since the longest p can be no longer than $|E|$. Giving a running time of $O(|f^*|\dot{(}E+E)) = O(E|f^*|)$.

Edmonds-Karp

Is a Ford-Fulkerson implementation that uses Shortest-Path to find the path p in line 3 of the Ford-Fulkerson algorithm. Each edge is given unit-weight and the algorithm will then pick the shortest path each time. It then has a running-time of $O(VE^2)$.

Lemma 26.7 If Edmonds-Karp is run on a flow network $G = (V, E)$ with source s and sink t, then for all vertices $v \in V - \{s, t\}$, the shortest-path distance $\delta_f(s, v)$ in the residual network G_f increases monotonically with each flow augmentation.

Proof-ish Intuitively, if we chose a path p from $s \to v$ that is a shortest path, and then assume there is a path p' from $s \to v$ which is shorter, we contradict our initial statement that p is a shortest path. This proves that the path length do not decrease. p' might have the same length as p or might be longer. $\square$

Theorem 26.8 If Edmonds-Karp is run on a flow network $G = (V, E)$ with source s and sink t, then the total number of flow augmentations performed by the algorithm is $O(VE)$.

Write down and understand the proof...

Fibonacci Heaps

Write something about mergable heaps

Fibonacci heaps are a datastructure that supports a set of operations qualifying it as a "meargeable heap", meaning it supports the following operations.

- `Make-Heap()` Creates and returns a new heap with no elements.
- `Insert(H,x)` Inserts element x whose key have already been filled in., into heap H.
- `Minimum(H)` Returns a pointer to the element in heap H whose key is minimum.
- `Extract-Min(H)` Deletes the element from heap H whose key is minimum, returning a pointer to the element.
- `Union(`H_1`,`H_2`)` Creates and returns a new heap that contains all the elements of both heaps. Both heaps are destroyed by the operation.

Apart from the meargeable heap operations above, Fibonacci Heaps also support the following two operations.

- `Decrease-Key(H,x,k)` Assigns to element x in heap H the new key value k, which cannot be greater than it's current value.
- `Delete(H,x)` Deletes element x from H.

Fibonacci Heaps are by default min-heaps, but could just as well be max heaps, then we would just replace the Minimum, Extract-Min and Decrease-Key operations with Maximum, Extract-Max and Increase-Key instead.

Fibonacci Heaps have a benefit in the fact that many operations are run in constat amortized time. So if these operations are used frequently, the Fibonacci Heap is a well suited structure.

Structure

A Fibonaccci Heap is a collectoin of rooted trees that are "min-heap ordered". That is, each tree obeys the minimum-heap property: The key of a node is greater than or equal to the key of its parent. A node x contains a pointer $x.p$ to its parent and a pointer $x.child$ to any one of its children. This list is called the

Procedure	Binary heap(worst case)	Fibonacci heap (amortized)
Make-heap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(lgn)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
Extract-Min	$\Theta(lgn)$	$O(lgn)$
Union	$\Theta(n)$	$\Theta(1)$
Decrease-Key	$\Theta(lgn)$	$\Theta(1)$
Delete	$\Theta(lgn)$	$O(lgn)$

Table 1: Amortized running times of normal binary heaps and Fibonacci Heaps.

child list. Each node also contains 2 pointers $x.left$ and $x.right$, these points to a nodes siblings or to the node itself if it has n osiblings. This forms a circular doubly-linked list called the child list. The nodes may appear in the child list in any order.

Nodes have 2 additional properties, $x.degree$ which is how many children a node have, and a boolean value $x.mark$. $x.mark$ indicates if x has lost a child since x was made the child of another node. Nodes initially have $x.mark$ = False.

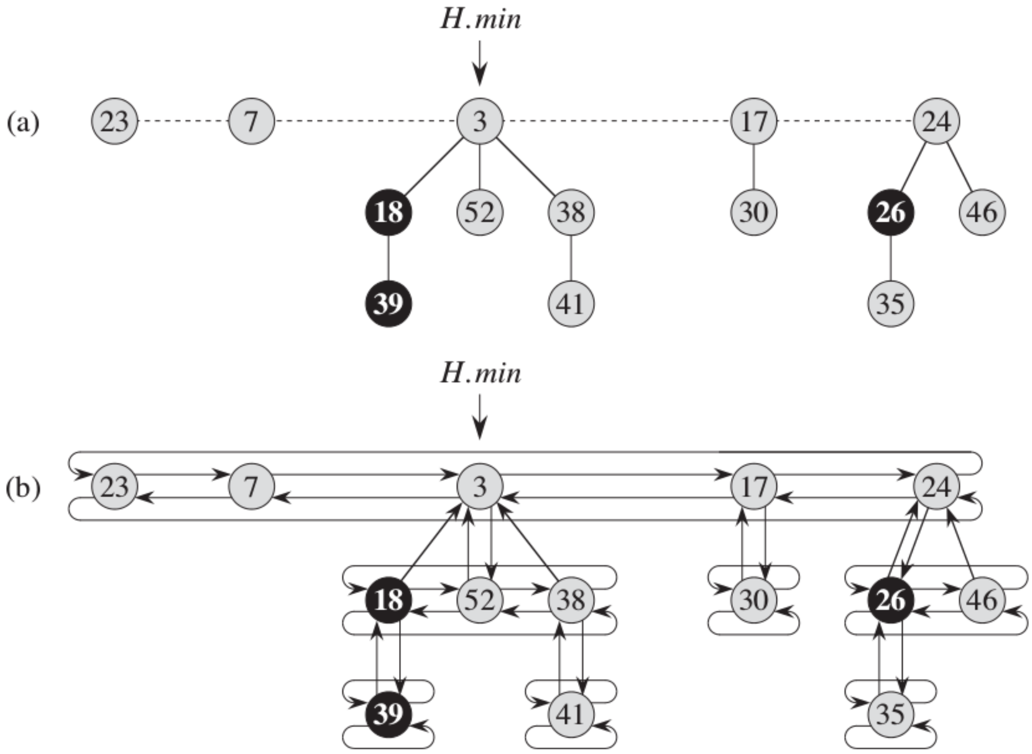

Figure 5: An example of a Fibonacci Heap with(b) and without(a) the child-list.

We access a fibonacci Heap by the pointer $H.min$ which points to the root of a tree containing the minimum key, this node is called the minimum node. If there are several nodes with the smallest key, any of those will work. When the heap is empty, $H.min$ = NIL. The roots of all the tree in a Fibonacci Heap ar linked togeter in a circular doubly-linked list called the "root list". $H.min$ points to a node in the root list. The heap also have one other property, $H.n$ which is the number of nodes currently in H.

Operations

Make-Heap Simply creates a pointer $H.min$ = NIL; since there is no trees in H at this point. This operation can be performed in $O(1)$ time.

Insert Insertion is done on constant time as well.

```
FIB-HEAP-INSERT(H, x)
x.degree = 1
x.p = NIL
x.child = NIL
x.mark = FALSE
if H.min == FALSE
    Create a root list for H containing just x.
    H.min = x
else insert x into H's root list
    if x.key < H.min.key
        H.min = x
H.n = H.n + 1
```

Minimum Just follow the $H.min$ pointer and you're home safe.

Extract-Min This is the most complicated and expensive of the meargeable heap procedures, since it will be doing the work of actually consolidating the trees in the list. Most other procedures put of this work so that it can be done when using Extract-Min. This operation can be done in $O(\lg n)$ time.

```
FIB-HEAP-EXTRACT-MIN(H)
z.min = H.min
if z ≠ NIL
    for each child x of z
        add x to the root list of H
        x.p = NIL
    remove z from the root list of H
    if z == z.right
        H.min = NIL
    else H.min = z.right
        CONSOLIDATE(H)
    H.n = H.n − 1
return z
```

Notice that $D(H.n)$ here will calculate the upper bound on the degree.

```
CONSOLIDATE(H)
let A[0..D(H.n)] be a new array
for i = 0 to D(H.n)
    A[i] = NIL
for each node w in the root list of H
    x = w
    d = x.degree
    while A[d] ≠ NIL
        y = A[d] // another node with the same degree as x
        if x.key > y.key
            exchange x with y
        FIB-HEAP-LINK(H, y, x)
        A[d] = NIL
        d = d + 1
    A[d] = x
H.min = NIL
for i = 0 to D(H.n)
    if A[i] ≠ NIL
        if H.min == NIL
            create a root list for H containing just A[i]
            H.min = A[i]
        else insert A[i] into H's root list
            if A[i].key < H.min.key
                H.min = A[i]
```

```
FIB-HEAP-LINK(H, y, x)
remove y from the root list of H
make y a child of x, incrementing x.degree
y.mark = FALSE
```

Union Merging two fibonacci Heaps are done in constant:

```
FIB-HEAP-UNION(H1, H2)
H = MAKE-FIB-HEAP()
H.min = H1.min
Concatenate the root list oh H2 with the root list of H.
if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
    H.min = H2.min
H.n = H1.n + H2.n
return H
```

DecreaseKey We can decrease the key of any node using this method, it runs in $O(1)$ time.

```
FIB-HEAP-DECREASE-KEY(H, x, k)
if k > x.key
    error new key is greater than current key"
x.key = k
y = x.p
if y ≠ NIL adn x.key < y.key
    CUT(H, x, y)
    CASCADING-CUT(H, y)
if x.key < H.min.key
    H.min = x
```

```
CUT(H, x, y)
remove x from the child list of y, decrementing y.degree
add x to the root list of H
x.p = NIL
x.mark = FALSE
```

```
CASCADING-CUT(H, y)
z = y.p
if z ≠ NIL
    if y.mark == FALSE
        y.mark = TRUE
    else CUT(H, y, z)
        CASCADING-CUT(H, z)
```

Delete Since it uses two previusly defined functions, the amortized running-time is easily calculated to $O(\lg n)$.

```
FIB-HEAP-DELETE(H, x)
FIB-HEAP-DECREASE-KEY(H, x, −∞)
FIB-HEAP-EXTRACT-MIN(H)
```

Wrwite something about $D(n)$ since it's used for all the proofs.

NP-Completeness

Example of a problem in the P class is an Euler Tour(a path in a graph that uses all edges exactly once, vertices can be visited multible times) of a graph, it can be done in $O(E)$. An NP class example that is very similar is a Hamiltonian cycle. A Hamiltonian cycle is a path that visits all vertices once.

We have three classes of problems in this subject:

P Problems that are solvable in polynomial time ($O(n^k)$ for some constant k.)

NP Problems that are verifiable on polynomial time, i.e. if we have a certificate/solution, can we check it in polynomial time. All problems in P will also be in NP.

NP-Hard A subclass of NPC problems that are "at least as hard as the hardest problems in NP", these cannot be verified in polynomial time.

NPC Problems that are both in the set of NP problems and NP-hard problems. (NP $\cap$ NP-Hard)

Decision problems vs. optimization problems

NP-completeness does not cover optimization problems, only decision problems. We can however use the relationship between optimization and decision problems to guage if a optimization problems is in fact NP-complete.

The shortest-path problem is an opimization problem, but can converted (in polynomial time) to a decision problem if the question is posed like so: "Does a path p in the graph G exist with only k edges?", then we iterate over k and will be able to guage the shortest bath problem as a dicision problem.

Reduction

The notion of showing that one problem us no harder or no easier than another problem applies even when both problems are decision problems. This is used in almost all NP-Completeness proof as follows: Take an instance α of problem A, that is, a specific input for the problem A, so for shortest path we may choose a graph G, and vertices u and v as well as a k. Make a polynomial time transofmration from α to instance β of problem B (which can be decided in polynomial time) with the following characteristics:

1. Transformation takes polynomial time.
2. The answers are the same. The answer for α is "yes", iff. the answer for β is "yes", same for "no".

Such an algorithm is called a reduction algorithm.

We can now solve any instance of A in polynomial time by converting α to β in polynomial time, running the polynomial time decision algorithm for B and using the answer for B as the answer for A.

Since NP-Completeness is usually how showing how hard a problem is we can use polynomial time reduction algorithms in the opposite way to show that a problem is NP-Ciomplete. Lets show that for some problem B there can be no polynomial time algorithm.

Suppose we have a dicision problem A, which we know that no polynomial time algorithm cat exist. Suppose we also have a polynomial time reduction algorithm that can reduce instances of A into instances of B instead. Now, suppose otherwise, if B had a polynomial time algorithm, we would be able to reduce A to B and have a polynomial time algorithm for A, which we assummed could not exist, which is a contradiction.

Abstract Problems

We define the abstract problem Q to be the binary relationship between an instance in the set I and a solution. For our shortest path problem, the instance would be a triple of input $i = (G, u, v)$ and the solution s would be a series of vertices. Since NP-Completenes is about decision problems, the input would instead be $i = (G, u, v, k)$ and the output would be $s = \{0, 1\}$. Resulting in a decicions problem like so $\text{PAth}(i) = 1$(yes) if a path exists in G between u and v using k vertices. and $\text{PAth}(i) = 0$(no) otherwise. We thus rely on the ability to recast optimization problems as decision problems in order to make decisions about their NP-Completeness.

Formal Language Framework

Σ is an alpjanet, a language L over the alphabet Σ is any combination of symbols from Σ. For instance if $\Sigma = \{0, 1\}$ we can have $L = \{0, 1, 10, 11, 101, 110, 111, ...\}$. We denote the empty string as ϵ and the empty language as $\emptyset$, and the language of all strings over Σ as Σ^*. For instance if $\Sigma = \{0, 1\}$ then $\Sigma^* = \{\epsilon, 0, 1, 10, 11, 100, 101, 110, 111, 1000, ...\}$. Every language Lover Σ is a subset of Σ^*.

We can perform several operations on languages, set operations such as union and intersect, or the complement of a lanaugege L as $\overline{L} = \Sigma^* - L$. The concatenation $L_1 L_2$ of two languages is

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}.$$

The closure or Kleene star of a language is

$$L^* = \{\epsilon\} \cup L \cup L^2 \cup L^3 \cup ...,$$

where L^k os the language obtained by concatenating L to itself k times.

Continue from p. 1058.

Randomized Algorithms

When talking random variables there are generally 3 kinds:

1. Las Vegas algorithms
2. Monte Carlo algorithms with one-sided error
3. Monte Carlo algorithms with two-sided error

A Las Vegas algorithm is a randomized algorithm that have zero possibility of producing an invalid solution but where the running time is affected vy the randomization.

A Monte Carlo algorithm is a randomized algorithm that might produce an incorrect solution. For decisions problems these can be one-sided or two-sided. A one sided algorithm is always correct for one of the answer (yes/no) but might be wrong on the other one. If it is two-sided then it might be wrong on both answers.

Randomized Min Cut

Events ε_1 and ε_2 are independent if the probability that they both occur are

$$\mathbf{Pr}[\varepsilon_1 \cap \varepsilon_2] = \mathbf{Pr}[\varepsilon_1] \times \mathbf{Pr}[\varepsilon_2] \tag{1}$$

When they are not necessarily independent,

$$\mathbf{Pr}[\varepsilon_1 \cap \varepsilon_2] = \mathbf{Pr}[\varepsilon_1|\varepsilon_2] \times \mathbf{Pr}[\varepsilon_2] = \mathbf{Pr}[\varepsilon_2|\varepsilon_1] \times \mathbf{Pr}[\varepsilon_1], \tag{2}$$

sometimes when a collection of events is not independent, a generalization of (2) can be used:

$$\mathbf{Pr}[\cap_{i=1}^{k} \varepsilon_i] = \mathbf{Pr}[\varepsilon_1] \times \mathbf{Pr}[\varepsilon_2|\varepsilon_1] \times \mathbf{Pr}[\varepsilon_3|\varepsilon_1 \cap \varepsilon_2] \ldots \mathbf{Pr}[\varepsilon_k| \cap_{i=1}^{k-1} \varepsilon_i]. \tag{3}$$

Consider a multigraph G (multigraphs can have multiple edges between two vertices). A cut in the graph $G$is the removal of a set of edges that divides the graph into two or more components. A min cut is a cut of minimum cardinality (cardinality is how many items are in a set). The algorithm works by contracting edges, contracting and edges simply means merge the two vertices at the end and retain all edges connected to them, delete any self-loops. This algorithm will work since all edges in the newly created graph will always be edges also in G so any cut in G' is also a cut in G.

The algorithm is simple:

1. Pick an edge uniformly at random and contract it to create G' with at least one edge less than G.
2. Repeat step 1 until only two vertices remain.
3. The edges between the two vertices in G' is output as a candidate min-cut.

Markov & Chebyshev's inequalities

Markov inequality Let Y be a random variable assuming only non-negative values. Then for all $t \in \mathbb{R}^+$,

$$\Pr[Y \geq t] \leq \frac{\mathrm{E[Y]}}{t}.$$

Equivalently,

$$\Pr[Y \geq k\mathrm{E}[Y]] \leq \frac{1}{k}.$$

Proof Define a function $f(y)$:

$$f(y) = \begin{cases} 1 & \text{iff } y \geq t \\ 0 & \text{otherwise.} \end{cases}$$

Then $\Pr[Y \geq t] = \mathrm{E}[f(y)]$. Since $f(y) \leq y/t$ for all y,

$$\mathrm{E}[f(Y)] \leq \mathrm{E}\left[\frac{Y}{t}\right] = \frac{\mathrm{E}[Y]}{t},$$

and the theorem follows. □

Chebyshevs inequality Let X be a random variable with expectation μ_X and a standard deviation of σ_X. Then for any $t \in \mathbb{R}^+$,

$$\Pr[|X - \mu_X| \geq t\sigma_X] \leq \frac{1}{t^2}$$

Proof Note that

$$\Pr[|X - \mu_X| \geq t\sigma_X] = \Pr[(X - \mu_X)^2 \geq t^2\sigma_X^2]$$

The random variable $Y = (X - \mu_X)^2$ has expectation σ_X^2, and applying the Markov inequality to Y bounds this probability from above by $1/t^2$. □

Hashing

Universal Hashing

We wish to generate a function $h : U \to [m]$ from a key universe U to a set of hash values $[m] = \{0, \ldots, m-1\}$. We want h to be universal, so that for any given distinct keys $x, y \in U$, when h is picked at random (independant from x and y), we have a low collision probability:

$$\Pr_h[h(x) = h(y)] \leq 1/m.$$

For many applications it suffices for some constant $c = O(1)$, we have

$$\Pr_h[h(x) = h(y)] \leq c/m.$$

Then h is called c-universal.

Simple Hash Tables w. Chaining

We have a set $S \subseteq U$ of keys that we wish to store and be able to retrieve in expected constant time. Let $n = |S|$ and $m \geq n$. We then pick a universal hash function $h : U \to [m]$, and create an array L of m lists/chains so that for $i \in [m]$, $L[i]$ is the list of keys that hash to i. To see if a key $x \in U$ is in S, we check if x is in the list $L[h(x)]$. This takes time propotional to $1 + |L[h(x)]|$. We add one for the constant time to look up the list, and then add the number of elements in the list itself since we need to walk through them.

If $x \notin S$ and h is universal, then the expected number of elements in $L[h(x)]$ is

$$E[|L[h(x)]|] = \sum_{y \in S} \Pr[h(y) = h(x)] = n/m \leq 1$$

Signature Hashes

Another application is to assign a unique signature $s(x)$ to each key. For this we want $s(x) \neq s(y)$ for all distinct keys $x, y \in S$. We pick a universal hash function $s : U \to [n^3]$. The probability of collision is calculated as

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

The first inequality is a “union bound”, the probability of that at least one of multiple events happen is at most the sum of their probabilities.

Multiply-mod-prime

Multiply-mod-prime is an implementation of a hashing function, note that if $m \geq u$ we can let h be the identity function, since we won't need randomness to avoid collision, so we assyme $m < u$.

Chose a prime number $p \geq u$. Uniformly random pick $a \in [p]_+ = \{1, ..., p-1\}$ and $b \in [p] = \{0, ..., p-1\}$, and define $h_{a,b}(x) : [u] \to [m]$ as

$$h_{a,b}(x) = ((ax + b) mod p) mod m) \tag{4}$$

Given distinct $x, y \in [u] \subseteq [p]$, we argue that for random a and b, the following is true,

$$\Pr_{a \in [p]_+, b \in [p]}[h_{a,b}(x) = h_{a,b}(y)] \leq 1/m. \tag{5}$$

For most of the proof we consider all $a \in [p]$, including $a = 0$. Ruling out $a = 0$, will be used in the end to get a tight bound from (5).

Fact 2.1 If p is prime and $\alpha, \beta \in [p]_+$ then $\alpha\beta \not\equiv 0 \pmod{p}$.

For a given pair $(a, b) \in [p]^2$, define $(q, r) \in [p]^2$ by

$$ax + b \mod p = q \tag{6}$$

$$ay + b \mod p = r \tag{7}$$

Lemma 2.2 Equations (6) and (7) define a 1-1 correspondence between pairs $(a, b) \in [p]^2$ and pairs $(r, q) \in [p]^2$.

Proof For a given pair $(q, r) \in [p]^2$, we will show that there is at most one pair $(a, b) \in [p]^2$, satisfying (6) and (7). Subtracting (6) from (7) modulo p, we get

$$(ay + b) - (ax + b) \equiv a(y - x) \equiv q - r \pmod{p}, \tag{8}$$

We claim that there is at most one a satisfying (8). Suppose that there is another a' satisfying (8). Subtracting the equations with a and a', we get

$$(a-a')(y-x) \equiv 0 (\mod p),$$

but since $a-a'$ and $y-x$ are both non-zero modulo p, this contradicts Fact 2.1. There is thus at most one a satisfying (8) for given (q,r). With this a, we need b to satisfy (6), and this determines b as

$$b = ax - q \mod p. \tag{9}$$

Thus, satisfying (6) and (7), each pair $(q,r) \in [p]^2$ thus corresponds to at most one pair $(a,b) \in [p]^2$. On the other hand, (6) and (7) define a unique pair $(r,q) \in [p]^2$ for each pair $(a,b) \in [p]^2$. We have p^2 pairs of each kind so the correspondence must be 1-1. □

Since $x \not\equiv y$, by Fact 2.1

$$r = q \Longleftrightarrow a = 0. \tag{10}$$

Thus we pick $(a,b) \in [p]_+ \times [p]$, we get $r \not\equiv q$.

Returning to the proof of (5), we get a collision if and only if $q \equiv r (\mod m)$, and we know that $q \neq r$. For a given r there is at most $\lceil p/m \rceil$ values of q with $q \equiv r (\mod m)$; namely r, $r+m$, 4r+2m 4, ... Ruling out $q = r$ leaves us at most $\lceil p/m \rceil - 1$ valus of q for each of the p values of r. Noting that

$$\lceil p/m \rceil - 1 \leq \frac{(p+m-1)}{m} - 1 = \frac{p-1}{m},$$

we get that the total number of collision pairs (q,r), $r \neq q$, is bounded by at most $p(p-1)/m$. Since each of the $(pp-1)$ pairs from $[p]_+ \times [p]$ are equally likely, we conclude that the collision probability is bounded by $1/m$, as required for universality.

Strong Universality

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

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

Observation 3.1 An equivelent difition of strong universality is that each key is hashed inoformly into $[m]$, and that distinct keys are hashed independently.

Proof Assuming strong universality and consider distinct keys $x, y \in U$. For any hash value $q \in [m]$, $\mathbf{Pr}[h(x) = q] = \sum_{r\in[m]} \mathbf{Pr}[h(x) = q \wedge h(y) = r] = m/m^2 = 1/m$, so $h(x)$ is uniform in $[m]$, and the same holds for $h(y)$. Moreover, for any hash value $r \in [m]$,

$$\begin{aligned}\mathbf{Pr}[h(x)=q|h(y)=r] &= \frac{\mathbf{Pr}[h(x)=q\wedge(y)=r]}{\mathbf{Pr}[h(y)=r]}\\ &= \frac{(1/m^2)}{(1/m)} = 1/m = \mathbf{Pr}[h(x)=q],\end{aligned}$$

so $h(x)$ is independent of $h(y)$. For the converse direction when $h(x)$ and $h(y)$ are independant,$\mathbf{Pr}[h(x) = q \wedge h(y) = r] = \mathbf{Pr}[h(x) = q]\cdot\mathbf{Pr}[h(y) = r]$ and when $h(x)$ and $h(y)$ are uniform, $\mathbf{Pr}[h(x) = q] = \mathbf{Pr}[h(y) = r] = 1/m$, so $\mathbf{Pr}[h(x) = q]\cdot\mathbf{Pr}[h(y) = r] = 1/m^2$. □

Storng universality can also be called 2-independence. We may also accept a relaxed notion of strong universality, we say a rnadom hash function $h :\rightarrow [m]$ is strongly s-universal if (1) every key x hash "close to" uniformly in the sense that for every hash value $q \in [m]$, we have $\mathbf{Pr}[h(x) = q] \leq c/m$, and (2) every pair of distinct keys hash independently.

Continue from page 7 of the hashing paper?

Exact Exponential Algorithms & Fixed-Parameter Tractable Problems

Exact Exponential Algorithms

The O^* notation is similar to the O notation, except it will suppress running-times of polynomial itme. This means that factors that are not exponential are suppressed. For example $O(kn^k k^n) = O(n^k k^n)$ but we have $O^*(kn^k k^n) = O^*(k^n)$.

Size vs length: When we talk about the running time we usually talk about the time in relation to the input size of length. For instance: Given a graph, the input size will be $O(V+E)$ while the length will be the number of bits it takes to encode the input with any reasonable encoding.

For the travelling salesman problem (er permutation problem) the input is a set of n cities, where we want to find the correct permutation. In this case candidate solutions are sets of n cities, of which there is $n!$. Thus the trivial algorithm runs in $O^*(n!)$.

Travelling Salesman using Dynamic Programming

For a subset of cities $S \subset \{1, 2, ..., n\}$ that includes 1, and $j \in S$, let $C(S, j)$ be the length of the shortest path visiting each node in S exactly once, starting at 1 and ending at j. When $|S| > 1$, we define $C(S, 1) = \infty$ since the path cannot start and end at 1.

We can express $C(S, j)$ in smaller sub-problems. We start at 1 and end at j; for the second to last city we have to pick some $i \in S$, so the overall path length is the distance from 1 to i; namely, $C(S - \{j\}, i)$ plus the length of the final edge d_{ij}. We pick the best such i:

$$C(S, j) = \min_{i \in S : i \neq j} C(S - \{j\}, i) + d_{ij}$$

The sub problems will be ordered by $|S|$.

```
ExactTSP({c_1, c_2, ...c_n}, d)
C({1}, 1) = 0
for s = 2 to n
    for all subsets S ⊂ {c_1, c_2, ...c_n} of sizes s and containing 1
        C(S, 1) = ∞
        for all j ∈ S, j ≠ 1
            C(S, j) = min{C(S − {j}, i) + d_ij : i ∈ S, i ≠ j}
return min_j C({1, ..., n}, j) + d_j1
```

There are at most $2^n n$ sub-problems, and each one takes linear time giving a final running time of $O(n^2 2^n)$ or $O^*(2^n)$.

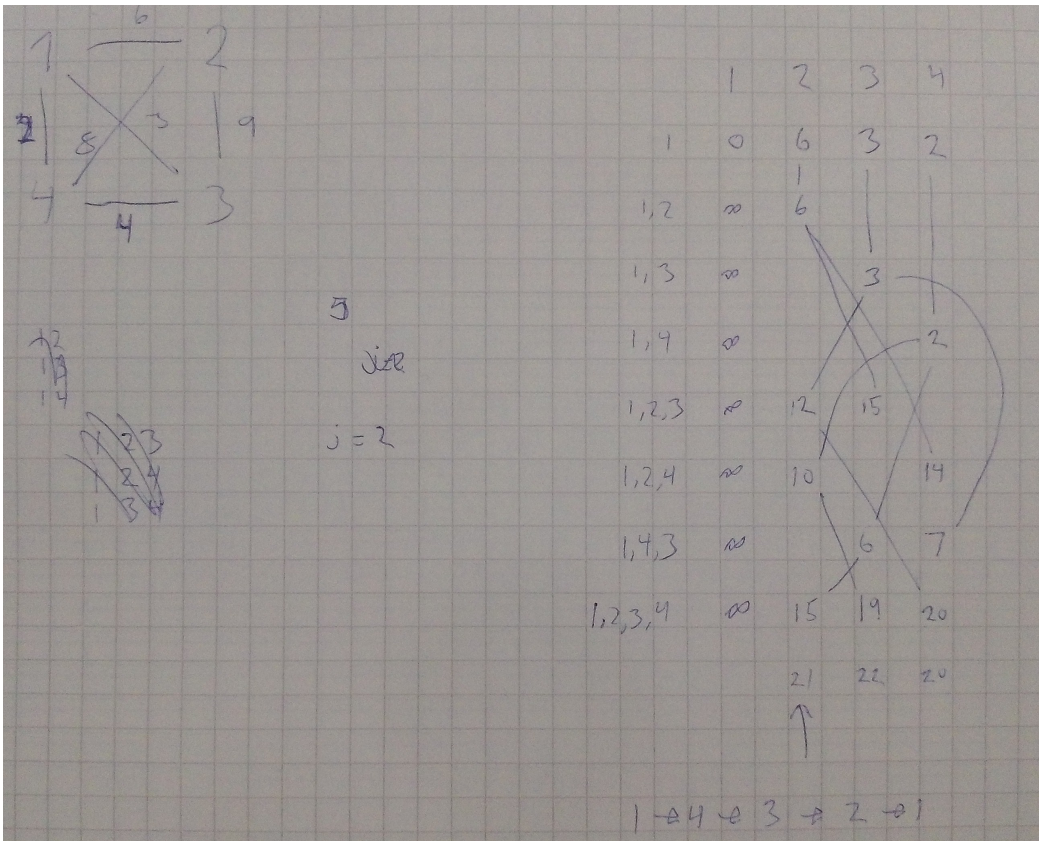

Figure 6: A run of the exact TSP algorithm

Fixed-Parameter Tractable Problems

Roughly speaking, parameterized complexity seeks the possibility of obtaining algorithms whose running time can be bounded by a polynomial function of the input length and, usually, an exponential function of a parameter which is independent of the input.

We strive to understand a problem and its sub-problems in terms of parameters and their effects on the running time, ideally the goal is to be able to form statements such as "If some parameter k is small in problem X then X can be solved efficiently". For instance in the vertex cover problem, we know that the solution should be as few vertices as possible, so if k is the solution size the exponential factor of the running time is less than 1.28^k.

CNF satisfiability

A CNF problem is a conjunction of m clauses where each clause consists of a disjunction of literals. There there be n different variables occurring in the formulae.

Parameter "Clause Size" The maximum number of k literals a clause may contain. For $k = 2$ (2-CNF satisfiability) the running time is polynomial time solvable, however for $k = 3$ (3-CNF satisfiability) is NP-Complete.

Parameter "Number of Variables" The number n of different variables allowed in the formula. Since there is essentially 2^n different truth assignments, the problem can be solved in that number of steps, seeing that the result of each assignment can be calculated in a number of steps equal to the Number of clauses.

Parameter "Number of Clauses" If the number of clauses in a formulae is bounded from above by m, the CNF problem can be solved in 1.24^m steps.

Parameter "Formula Length" If the total length (counting the number of literal occurrences in the formula) of the formula F is bounded by above by $\ell = |F|$, then the problem can be solved in 1.08^ℓ steps.

Maximum CNF Satisfiability

An optimization version of the above problem. Here we want to satisfy as many clauses as possible and not just the entire formulae. This problem shares the same parameters, but gives different effects.

Parameter "Clause Size" Even for $k = 2$ the problem is still NP-Complete.

Parameter "Number of Variables" Like the decision problem, this one can be solved in 2^n steps, where n is the number of variables in the formulae. For $k = 3$ it is still an open problem to obtain a better bound. Recently advances were made for $k = 2$ allowing it to be solved in 1.74^n steps.

Parameter "Number of Clauses" IFw e can bound the number of clauses above by m, the Maximum Satisfiability problem can be solved in 1.33^m steps, and the Maximum 2 Satisfiability problem can be solved in 1.15^m steps.

Parameter "Formula Length" If the total length (counting the number of literal occurrences in the formula) of the formula F is bounded by above by ℓ, Maximum Satisfiability problem can be solved in 1.11^ℓ steps, and the Maximum 2 Satisfiability problem can be solved in 1.08^ℓ steps.

VRÆÆÆÆÆÆÆÆLLLLL :(

Approximation Algorithms

Performance Ratios and Schemes

An algorithm for a problem have an approximation ratio of $\rho(n)$ if, for any input of size n, the cost C of a solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

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

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-approximation algoroithm. These notions apply to both cost-minimization and cost-maximization problems.

For a miximization problem, $0 < C \leq C^*$, and the ratio C^*/C gives the factor by which the cost of an optimal solution is larger than the cost of the approxmiation solution.

Similarly, for a minimization problem, $0 < C^* \leq C$, and the ratio C/C^* gives the factor by which the cost of the approximate solution is larger than the cost of an optimal solution.

Because we assume the costs are allways positive, the rations are always well defined, the ratio of an algorithm is never less than 1, since $C/C^* \leq 1$ implies $C^*/C \geq 1$. Therefore a 1-approximation algorithm produce an optimal solition and an approximation algorithm with a large approximation ratio may return a solution far worse than optimal.

Below some schemes are outlined which allow us to give a value ϵ along with the instance of the problem and achieve an approximation that have a quality depending on the value.

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

Polynomial-Time Approximation Scheme is an approximation scheme if for any fixed $\epsilon > 0$, the scheme runs in polynomia ltime in the size n of it's input instance. The running time of such a scheme can increase rapidly as

ϵ decreases. For example, the running time of a polynomial-time approximation scheme might be $O(n^{(2/\epsilon)})$. Ideally, if ϵ decrease by a constant factor, the running time to achieve the desired approximation should not incrase by more than a constant factor. (Not necessarily by the same factor ϵ was decreased with.)

Fully Polynomial-Time Approximation Scheme means the approximation algorithm runs in polynomial time in both $1/\epsilon$ and the size n of the input instance. I.e. $O((1/\epsilon)^2 n^3)$. With such a scheme, a constant factor decrease in ϵ comes with a constant factor increase in runningtime.

Vertex Cover example

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, this is an optimization version of an NP-Complete decision problem.

An approximation algorithm that will find a cover that is no more than twice the size of the optimal cover is written here:

APPROX-VERTEX-COVER(G)

$C = \emptyset$
$E' == G.E$
while $E' \neq \emptyset$
    let (u, v) be an arbitrary edge of E'
    $C = C \cup \{u, v\}$
    remove from E' every edge incident on either u or v
return C

The above algorithm runs in $O(V + E)$.

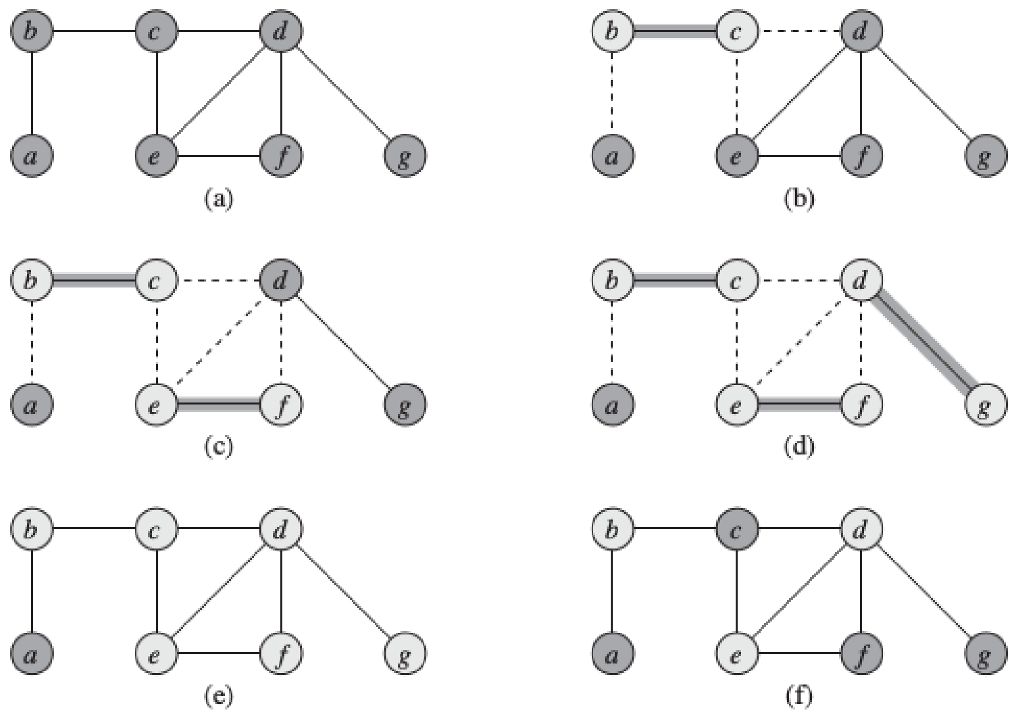

Figure 7: Example run of the `Approx-Vertex-Cover` algorithm.

Theorem 35.1 `Approx-Vertex-Cover` is a polynomial-time 2-approximation algorithm.

Proof It is already shown that the algorithm is a polynomial time algorithm.

The set C of vertices return must be a vertex cover since it loops until every edge in $G.E$ have been covered by some vertex.

To see that the algorithm returns a vertex cover of at most twice the size of an optimal cover, let A denote the set of edges that line 4 selects. To cover all the edges in A, any vertex cover, in particular the optimal vertex cover C^*, must include at least one endpoint of each edge in A. No two edges share endpoints since when an edge is picked, all edges incident on its endpoints are removed. Thus no two edges in A are covered by the same vertex from C^*, and we have the lower bound

$$|C^*| \geq |A|$$

on the size of an optimal vertex cover. Each execution of line 4 picks an edge which neither of its endpoints are already in C, yielding an upper boind on the size of the vertex cover returned

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

Combining the two bounds gives

$$\begin{aligned} |C| &= 2|A| \\ &\leq 2|C*| \end{aligned}$$

thereby proving the theorem. □

Traveling Salesman Problem

The TSP is also NP-Complete, the following is a method for approximating an optimal solution that is at most twice as long as the optimal tour.

It creates a minimum spanning tree using Prims algorithm(see below), and walks along this tree using a pre-order walk (parent first then child), and uses this as the solution.

```
APPROX-TSP-TOUR(G, c)
    select a vertex r ∈ G.V to be a "root" vertex
    compute a minimum spanning tree T for G from root r
        using MST-PRIM(G, c, r)
    let H be a list of vertices, ordered according to when
        they are first visited in a preorder tree walk of T
    return the hamiltonian cycle H
```

Q is a min-priority queue, G is the graph, r is the root and w is the weight function. (It basically keeps adding the cheapest edge it can find.)

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NUL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u.v)
```

The `Approx-TSP-Tour` Algorithm runs in $\Theta(V^2)$. It's tightly bound by the inner loop in MST-Prim.

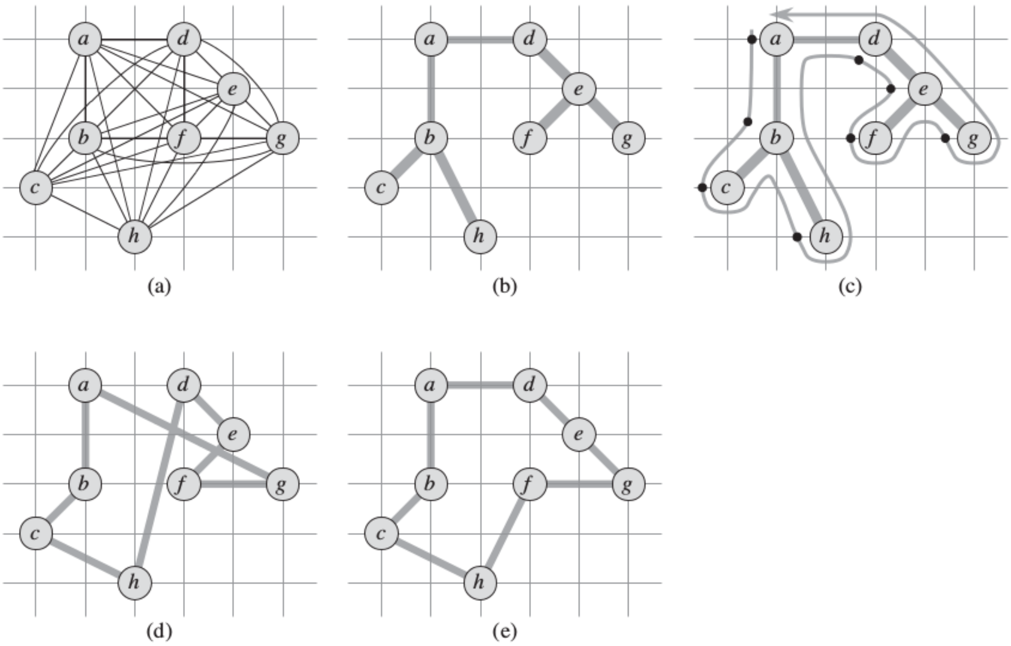

Figure 8: A sample run of the approximate TSP algorithm, subigure (d) is the resulting tour, and (e) is the optimal tour. Distances are simple euclidian distances.

Theorem 35.2 `Approx-TSP-Tour` is a polynomial-time 2-approximation algorithm for the traveling salesman problem, with the triangle inequality.

Proof We've already seen that the algorithm runs on polynomial time. LEt H^* denote an optimal tour of a given set of vertices. We obtain a spanning tree be deleting any edge from a tour and each edge cost is nonnegative. The weight of the MST T calculated in line 2 of `Approx-TSP-Tour` is a lower bound for the cost of an optimal tour:

$$c(T) \leq c(H^*) \tag{11}$$

A full walk list a vertex when it's first visited and on all subsequent revisits, lets call a full walk of our tree W. Since the full walk traverses every edge of T exactly twice, we have

$$c(W) = 2c(T) \tag{12}$$

Combinging Equation 11 and 12 imply that

$$c(W) \leq 2c(H^*) \tag{13}$$

and so the cost of W is within a factor of 2 of the cost of an optimal tour. A full walk of T is usually not a tour though, since it might visit some vertices several times. By the triangle inequality, we can delete any such revisit though without increasing the cost. Doing this for all revisits gives a set of vertices corresponding to the-preorder walk of T. Let H be the cycle that describes the preorder walk. It is a hamiltonian cycle since every vertex is visited once, and it is the cycle computed by `Approx-TSP-Tour`. Since H is created be removing vertices from W we have

$$c(H) \leq c(W) \tag{14}$$

combining 13 and 14, gives $c(H) \leq 2c(H^*)$, which completes the proof. □

If we drop the assumption that the cost function c satisfies the triangle inequality, we cannot approximate a tour in polynomial-time unless $P = NP$.

Theorem 35.3 If $P \neq NP$, then for any constant $\rho \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio ρ for the general traveling salesman problem.

Proof This will be a proof by contradiction. Suppose to the contrary that for some number $\rho \geq 1$, there is a polynomial-time approximation algorithm A with approximation ratio ρ. Without loss of generality we assume that ρ is an integer, by rounding up if necessary. We shall then show how to use A to solve instances of the Hamiltonian-cycle problem in polynomial time. Since the Hamiltonian-cycle problem is NP-Complete a solution to the problem would mean $P = NP$.

Let $G = (V, E)$ be an instance of the Hamiltonian-cycle problem. We wish to determine efficiently whether G contains a Hamiltonian cycle by making use of the hypothesized approximation algorithm A. We turn G into an instance of the traveling-salesman problem as follows. Let $G' = (V, E')$ be a complete graph on V; that is,

$$E' = \{(u, v) : u, v \in V \text{ and } u \neq v\}.$$

Assign an integer cost to each edge in E' as follows:

$$c(u, v) = \begin{cases} 1 & \text{if } (u, v) \in E, \\ \rho|V| + 1 & otherwise. \end{cases}$$

We can create representations of G' and c from a representation of G in time polynomial in $|V|$ and $|E|$.

Now consider the traveling-salesman problem (G', c). If the original graph G has a hamiltonian cycle H, then the cost function c assigns to each edge of H a cost of 1, and so (G', c) contains a tour of cost $|V|$. On the other hand, if G does not contain a hamiltonian cycle, then any tour of G' must use some edge not in E. But any tour that uses an edge not in E has a cost of at least

$$\begin{aligned} (\rho|V| + 1) + (|V| - 1) &= \rho|V| + |V| \\ &> \rho|V|. \end{aligned}$$

Because edges not in G are so costly, there s a gap of at least $\rho|V|$ between the cost of a tour that is a hamiltonian cycle in G (cost $|V|$) and the cost of any other tour (cost at least $\rho|V| + |V|$). Therefore, the cost of a tour that is not a hamiltonian cyclein G is at least a factor of $\rho + 1$ greater than the cost of a tour that is a hamiltonian cycle in G.

Now, suppose that we apply the approximation algorithm A to the traveling-salesman problem (G', c). Because A is guaranteed to return a tour of cost no more than ρ times the cost of an optimal tour, if G contains a hamiltonian cycle, then A must return it. If G has no hamiltonian cycle, then A returns a tour of cost more than $\rho|V|$. THerefore, we can use A to solve the hamiltonian-cycle problem, in polynomial time. $\square$

Computational Geometry

Let $P = \{p_1, p_2, ..., p_n\}$ be a set of points in the plane. To be able to properly define the triangulation of the plane we first define the "maximal planar subdivision" as a subdivision $\mathscr{S}$ such that no edges that connects two vertices can be added without destroying the planarity.

A triangulation of P is now defined as a maximal planar subdivision whose vertex set is P.

Theorem Let P be a set of n points in the plane, not all collinear, and let k denote the number of points in P that lie on the boundary of the convex hull of P. Then any triangulation of P has $2n-2-k$ triangles and $3n-3-k$ edges.

Proof Let $\mathscr{T}$ be a triangulation of P, and let m denote the number of triangles of $\mathscr{T}$. Note that the number of faces of the triangulation, which we denote by n_f, is $m+1$. Every triangle has three edges, and the unbounded face has k edges. Furthermore, every edge is incident to exactly two faces. Hence the total number of edges of $\mathscr{T}$ is $n_e = (3m+k)/2$. Euler's formula tells us that

$$n - n_e + n_f = 2$$

Plugging the values for n_e and n_f into the formula, we get $m = 2n-2-k$, which in turn implies $n_e = 3n-3-k$. $\square$

Let $\mathscr{T}$ be a triangulation of P, and suppose it has m triangles. Consider the $3m$ angles of $\mathscr{T}$, sorted by increasing value. Let $\alpha_1, \alpha_2, ..., \alpha_{3m}$ be the resulting sequence of angles. We call $A(\mathscr{T}) = (\alpha_1, \alpha_2, ..., \alpha_{3m})$ the angle vector of $\mathscr{T}$. Let $\mathscr{T}'$ be another triangulation of the same point set P. We say that $A(\mathscr{T}) > A(\mathscr{T}')$ if $A(\mathscr{T})$ if there exists and index i with $1 \le i \le 3m$ such that

$$\alpha_j = \alpha'_j \text{ for all } j < i, \quad \text{and} \quad \alpha_i > \alpha'_i$$

A triangulation $\mathscr{T}$ is called angle-optimal if $A(\mathscr{T}) \ge A(\mathscr{T}')$ for all triangulations $\mathscr{T}'$ of P. These are good because slender triangles make for bad triangulations for terrain.

Denote the smaller angle defined by three points p, q, r, as $\measuredangle pqr$.

Thales Theorem Let Cbe a circle, ℓ be a line intersecting C in points a and b. Let p, q, r and s be points lying on the same side fo ℓ. Suppose that p and q lie on C, that r lies inside C nad that s lies outside C. Then

$$\measuredangle arb > \measuredangle apb = \measuredangle aqb > \measuredangle asb.$$

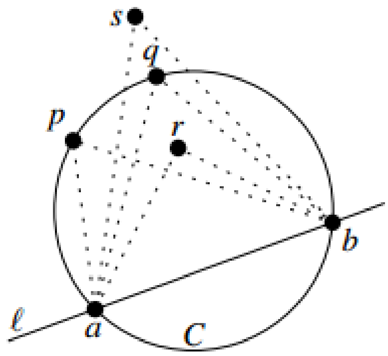

Figure 9: The circle C and the points drwan for clarity.

Illegal edge Consider an edge $e = \overline{p_i p_j}$ of a triangulation $\mathcal{T}$. If e is not on the unbounded face, it is incident on two triangles, $p_i p_j p_k$ and $p_i p_j p_l$. If these triangles form a convex quadrilateral, we can obtain a new triangulation $\mathcal{T}'$ by flipping the edge. This is done by removing $\overline{p_i p_j}$ and adding $\overline{p_k p_l}$. This changes the anglevectors, but only the entries associated with the two triangles. An edgeis considered illegal if flipping it increase $A(\mathcal{T})$ such that

$$\min_{1 \le i \le 6} \alpha_i < \min_{1 \le i \le 6} \alpha'_i.$$

An edge is illegal if we can locally increase the smallest angle simply by flipping it.

Observation 9.3 Let $\mathcal{T}$ be a triangulation with an illegal edge e. Let $\mathcal{T}'$ be the triangulation obtained from $\mathcal{T}$ by flipping e. Then $A(\mathcal{T}') > A(\mathcal{T})$.

Lemma 9.4 Let edge $\overline{p_i p_j}$ be incident on to triangles $p_i p_j p_k$ and $p_i p_j p_k$, and let C be the circle through p_i, p_j and p_k. The dege $\overline{p_i p_j}$ is illegal iff. the point p_l lies in the interior of C. Furthermore, if the points p_i, p_j, p_k and

p_l form a convex quadrilateral and do not lie on a common circle, then exactly one of $\overline{p_i p_j}$ and $\overline{p_k p_l}$ is an illegal edge.

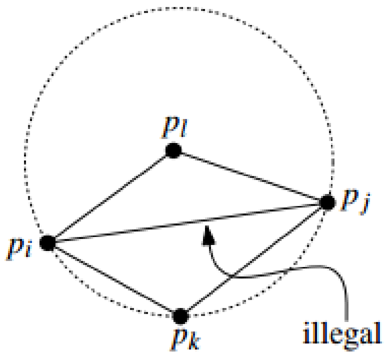

Figure 10: Visualization of Lemma 9.4.

Legal Triangulation A legal triangulation is simple a triangulation that contain no illegal edges. We note that any legal triangulation must also be an angle-optimal triangulation. Computing a legal triangulation is simple given any triangulation. simply flip illegal edges until non remain. This is very slow though.

Delaunay Triangulation

DELAUNAYTRIANGULATION(P)

Let p_0 be the lexicographically higest point og P (largest x and y coordinate.)
Let p_{-1} and p_{-2} be two points in $\mathbb{R}^2$ sufficiently far away such that P is contained in the triangle $p_0 p_{-1} p_{-2}$
Initialize $\mathcal{T}$ as the triangulation consisting of $p_0 p_{-1} p_{-2}$.
Compute a random permutation $p_1, p_2, \ldots, p_n$ of $P \backslash \{p_0\}$.
for $r = 1$ **to** n
    // Insert p_r into $\mathcal{T}$
    Find a triangle $p_i p_j p_k \in \mathcal{T}$ containing p_r
    if p_r lies on the interior of the triangle
        Add edges from p_r to the three vertices $p_i p_j p_k$, splitting it into three triangles.
        LEGALIZEEDGE($p_r, \overline{p_i p_j}, \mathcal{T}$)
        LEGALIZEEDGE($p_r, \overline{p_j p_k}, \mathcal{T}$)
        LEGALIZEEDGE($p_r, \overline{p_k p_i}, \mathcal{T}$)
    else // p_r must lie on an edge, say $\overline{p_i p_j}$
        Add edges from p_r to p_k and to the third vertex p_l of the other triangles that is incident to $\overline{p_i p_j}$, thereby splitting the two triangles incident to $\overline{p_i p_j}$ into four triangles.
        LEGALIZEEDGE($p_r, \overline{p_i p_l}, \mathcal{T}$)
        LEGALIZEEDGE($p_r, \overline{p_l p_j}, \mathcal{T}$)
        LEGALIZEEDGE($p_r, \overline{p_j p_k}, \mathcal{T}$)
        LEGALIZEEDGE($p_r, \overline{p_k p_i}, \mathcal{T}$)
Discard p_{-1} and p_{-2} and all their incident edges from $\mathcal{T}$.
return $\mathcal{T}$

LEGALIZEEDGE($p_r, \overline{p_i p_j}, \mathcal{T}$)

if $\overline{p_i p_j}$ is illegal
    Let $p_i p_j p_k$ be the truangle adjacent to $p_r p_i p_j$ along $\overline{p_i p_j}$.
    Replace $\overline{p_i p_j}$ with $\overline{p_r p_k}$.
    LEGALIZEEDGE($p_r, \overline{p_i p_k}, \mathcal{T}$)
    LEGALIZEEDGE($p_r, \overline{p_k p_j}, \mathcal{T}$)

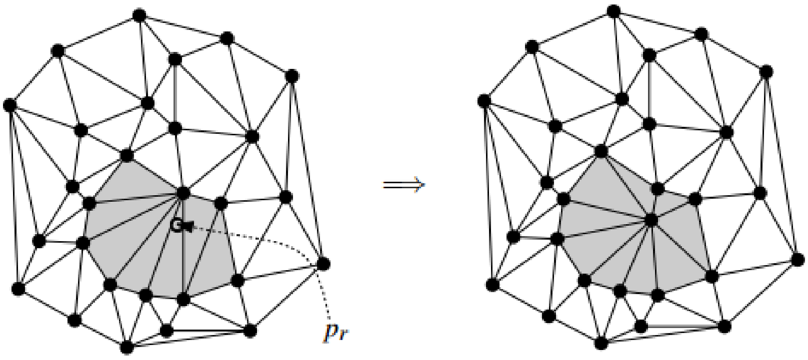

Figure 11: An example insertion of p_r and a calls to legalize edge.

Lemma 9.10 Every new edge created in `DelaunayTriangulation` or in `LegalizeEdge` during the insertion of p_r, is an edge of the Delaunay graph of $\{p_{-2}, p_{-1}, p_0, ..., p_r\}$

Proof Consider first the edges $\overline{p_r p_i}$, $\overline{p_r p_j}$,$\overline{p_r p_k}$ (and perhaps $\overline{p_r p_l}$) created by splitting $p_i p_j p_k$ (and maybe $p_i p_j p_l$). Since $p_i p_j p_k$ is a triangle in the Delaunay triangulation before the addition of p_r, the circumcirle C of $p_i p_j p_k$ contains no point p_t with $t < r$ in its interior. By shrinking C we can find a circle C' through p_i and p_r contained in C. Because $C' \subset C$ we know that C' is empty. This implies that $\overline{p_r p_i}$ is an edge of the Delaunay graph after the addition of p_r. The same holds for $\overline{p_r p_j}$ and $\overline{p_r p_k}$ (and $\overline{p_r p_l}$ if it exists).

Now consider and edge flipped by `LegalizeEdge`. Such an edge flip always replaces an edge $\overline{p_i p_j}$ of a triangle $p_i p_j p_l$ by and edge $\overline{p_r p_l}$ incident to p_r. Since $p_i p_j p_l$ was a Delaunay triangle before the addition of p_r and becauseits circumcircle C contains p_r – otherwise $\overline{p_i p_j}$ would not be illegal – we can shrink the circumcircle to obtain an empty circle C' with only p_r and p_l on its boundary. Hence $\overline{p_r p_l}$ is an edge of the Delaunay graph after the addition $\square$

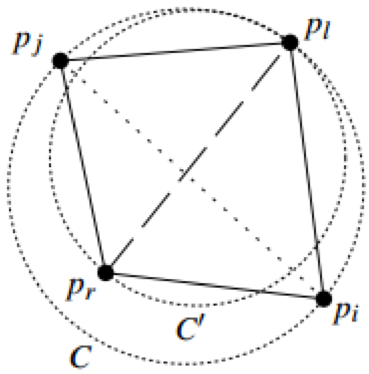

Figure 12: Helpfull illustration for the proof. :)

Look op the stuff on p. 202 of the comp geo paper, it's prett smart with pointers and trees.