Advanced Algorithms: Notes

Author: Martin Grünbaum (martin@itsolveonline.net)

June 18, 2014

Contents

1 Max-flow: Disposition

1. Flow network G
2. Flow $|f|$ (Capacity constraint, flow conservation)
3. Residual network, augmenting paths, cuts.
4. Max-flow min-cut theorem
5. Ford-Fulkerson method
6. Edmonds-Karp

CLRS 26.1, 26.2, 26.3

2 Max-flow: Notes

A flow network $G = (V, E)$ is a directed graph where each edge $(u, v) \in E$ has a non-negative capacity $c(u, v) \geq 0$. If there is an edge $(u, v) \in E$ then there is no edge $(v, u) \in E$. If $(u, v) \notin E$ then $c(u, v) = 0$ for convenience. When $(u, v) \notin E$, $f(u, v) = 0$.

Flow networks have a source s and a sink t. For each vertex $v \in V$, the flow network contains a path $s \rightsquigarrow v \rightsquigarrow t$. The graph is therefore connected, meaning $|E| \geq |V| - 1$.

A flow is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies two properties:

Capacity constraint: For all $u, v \in V$, $0 \leq f(u, v) \leq c(u, v)$

Flow conservation: For all $u \in V - \{s, t\}$, $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$.

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

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

In the **maximum-flow** problem, we are given a flow network G and we wish to find a maximum flow.

Edges are anti-parallel if there is both an edge (u, v) and an edge (v, u). This is not allowed, and to get around this we instead introduce a new edge x and re-structure the edges as follows: $(u, x), (x, v), (v, u)$. The capacity of the new edges involving x is the same as the capacity from (u, v). See page 711 in the book for an example.

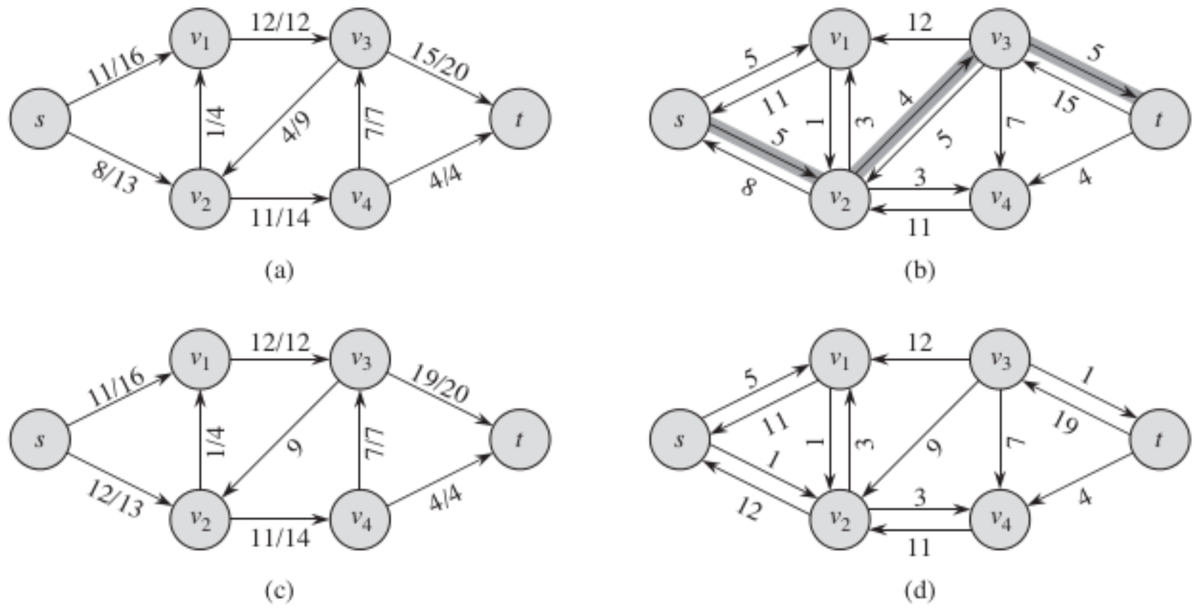
2.1 Multiple sources and sinks

This can be accounted for by introducing a **supersink** and **supersource** with infinite flow and capacity out to all of the sources and from all of the sinks to the supersink. See page 713.

2.2 Ford-Fulkerson method

Three basic principles: **residual networks**, **augmenting paths** and **cuts**. Essential for **max-flow min-cut** theorem (Theorem 26.6).

Intuition is as follows: We have a flow network G. We iteratively alter the flow of G, by finding an augmenting path in an associated residual network G_f. Once we know the edges that belong to an augmenting path, we can identify specific edges in G to increase or decrease the flow of. Each iteration



increases overall flow, but it may do so by decreasing the flow along certain edges. This is repeated until the residual network G_f has no more augmenting paths.

max-flow min-cut shows that upon termination, this yields a maximum flow.

2.2.1 Residual network

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

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

Residual capacity $c_f(u, v)$ is defined by

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

Note: that $(u, v) \in E$ implies $(v, u) \notin E$, so there is always only one of the three above cases that applies.

Because the edges in E_f are either edges from E or an edge in the opposite direction, $|E_f| \leq 2|E|$.

Intuition: A residual network G_f consists of edges with capacities that represent how we can alter the flow on edges of G. G can admit an additional amount of flow along an edge, equal to the capacity minus the current flow. If the edge can admit more flow, that edge is placed into G_f with a value of $c_f(u, v) = c(u, v) - f(u, v)$. The residual network may also contain edges that are not in G: In order to represent a possible decrease of a flow $f(u, v)$ on an edge in G, we place an edge (v, u) into G_f with residual capacity $c_f(v, u) = f(u, v)$. In other words, an edge that can admit flow in the opposite direction, at most cancelling out flow entirely. See Figure **??** for an example.

Flows in a residual network satisfy the definition of a flow, but with respect to capacities c_f in the network G_f. If f is a flow in G and f' is a flow in the corresponding residual network G_f, we define $f \uparrow f'$, the **augmentation flow** of f by f', as a function from $V \times V$ to $\mathbb{R}$ defined by

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

Intuition: Increase the flow ($f(u,v)$) by $f'(u,v)$, but decrease it by the flow in the opposite direction ($f'(v,u)$). Pushing flow in the reverse direction is also called **cancellation**.

2.2.2 Augmenting path

An augmenting path p is a simple path from s to t in the residual network G_f. By the definition of a residual network, we may increase the flow of an edge (u,v) by up to $c_f(u,v)$ without violating the capacity constraint on whichever of (u,v) and (v,u) is in the original flow network G.

The maximum amount by which we can increase flow on each edge of an augmenting path p is the **residual capacity** of p, given by $c_f(p) = min\{c_f(u,v) : (u,v) \text{ is on p}\}$. More specifically, if p is an augmenting path in G_f, we define a function $f_p : V \times V \to \mathbb{R}$ as

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

Then f_p is a flow in G_f with value $|f_p| = c_f(p) > 0$. See Lemma 26.2, page 720. It remains to be shown that augmenting f by f_p produces a different flow in G whose value is closer to the maximum. Corollary 26.3 on page 720 shows this by immediate proof, using Lemma 26.1 and 26.2.

2.2.3 Cuts of a network

We know, based on the above, that we can augment flows in G and that doing so can produce a new flow closer to the maximum. But how do we know that when it terminates, the algorithm has in fact found a maximum flow? Max-flow min- cut tells us that a flow is maximum only if its residual network contains no augmenting paths.

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 is 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

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

Intuitively, the capacity of the cut is the capacity of all vertices going from S to T, while the flow is the flow of vertices going from S to T, minus the flow going from T to S. A **minimum cut** of a network is a cut whose capacity is minimum over all cuts of the network.

Theorem 26.6 (Max-flow min-cut theorem, p. 723/724) involves proving the equivalence of 3 different conditions:

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

1 => 2: Assume that f were a maximum flow in G and there **was** an augmenting path. This means, by the proof of augmenting paths, that we could create a new flow f' in G with a strictly larger flow value than f, i.e. that $|f'| > |f|$. This contradicts f being a maximum flow.

2 => 3: Suppose that there are no augmenting paths, that is there is no path from s to t in G_f.

Define $S = \{v \in V : \text{there exists a path from } s \text{ to } v \text{ in } G_f\}$. That is, the set S contains all those vertices for which there could be pushed more flow along, but which perhaps have not because a later capacity limits that possibility. Define $T = V - S$. A partition (S, T) is a cut, where $s \in S$ and $t \notin S$ (since there is no path from s to t, or we would not have a maximum flow).

Consider two vertices (u, v) where $u \in S$ and $v \in T$:

If $(u, v) \in E$, we must have that $f(u, v) = c(u, v)$. If this were not the case we would have $(u, v) \in E_f$, since we would be able to push more flow out until at capacity. Then, by the definition of S we would have that $v \in S$. This is a contradiction.

If $(v, u) \in E$, we must have that $f(v, u) = 0$. If this were not the case we would have $(v, u) \in E_f$, since the residual capacity $c_f(u, v) = f(v, u)$ would be positive. This means $(u, v) \in E_f$, and we would have that $v \in S$. This is a contradiction.

Therefore:

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

3 => 1: The value of **any** flow f in a flow network G is bounded from above by the capacity of **any** cut of G. Proof:

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

Because of this, $|f| \leq c(S, T)$ for **all** cuts (S, T). Therefore, $|f| = c(S, T)$ implies that f is a maximum flow.

Algorithm 1 Ford-Fulkerson Method

```
1: procedure FORD-FULKERSON METHOD(G,s,t)
2:     Initialize flow $f$ to 0
3:     while there exists an augmenting path $p$ in residual network $G_f$ do
4:         Augment flow $f$ along $p$
5:     end while
6: end procedure
```

Running time of the simple algorithm depends on how we find the augmenting path p. If we assume an appropriate data structure where we can represent the directed graph, the time to find an appropriate path can be linear in the number of edges, if using breadth-first or depth-first search.

This gives us $O(E)$ work per iteration of the while loop, and at most the same number of iterations as the value of the maximum flow (since we increase by at least one unit per iteration). The total running time is therefore $O(|f*|E)$, where $|f*|$ is the maximum flow.

Algorithm 2 Ford-Fulkerson Basic Algorithm

```
procedure FordFulkersonSimple(G,s,t)
    for each edge $(u, v) \in G.E$ do
        $(u, v).f = 0$
    end for
    while there exists a path $p$ from $s$ to $t$ in residual network $G_f$ do
        $c_f(p) = \min\{c_f(u, v) : (u, v)$is in $p\}$
        for each edge $(u, v)$ in $p$ do
            if $(u, v) \in G.E$ then
                $(u, v).f = (u, v).f + c_f(p)$
            else
                $(v, u).f = (v, u).f - c_f(p)$
            end if
        end for
    end while
end procedure
```

2.3 Edmonds-Karp

Edmonds-Karp works by finding the shortest augmenting path each time. We choose the augmenting path as a shortest path from s to t in the residual network, where each edge has unit distance (weight). Edmonds-Karp runs in $O(VE^2)$ time.

(Lemma 26.7) If the Edmonds-Karp algorithm 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.

(Theorem 26.8) If the Edmonds-Karp algorithm 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 $O(VE^2)$

Each iteration of Edmonds-Karp, at least one edge along the augmenting path is a critical edge (an edge that bottlenecks the path, i.e. an edge that becomes saturated). The main idea behind the theorem is to show that each edge can become critical at most $|V|/2$ times. And since there are $O(E)$ pairs of vertices that can have an edge between them in a residual network, the total number of critical edges during the entire execution of Edmonds-Karp is $O(VE)$.

Using breadth-first search to find an augmenting path in $O(E)$ time, this gives a total running time of $O(VE^2)$.

3 Fibonacci heaps: Disposition

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

- Delete

CLRS 19

4 Fibonacci heaps: Notes

4.1 Mergeable heaps

A mergeable heap is one which supports the following operations:

Make-Heap() Creates and returns a new heap containing no elements.

Insert(H,x) Inserts element x, whose key has 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 heaps H_1 and H_2. The original heaps are destroyed by this operation.

In addition to the above operations, Fibonacci heaps also support the following two operations:

Decrease-Key(H,x,k) Assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.

Delete(H,x) Deletes element x from heap H.

Table 1: Running time of operations

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

Fibonacci heaps are theoretically desirable when the number of Extract-Min and Delete operations is small relative to the number of other operations performed. This can be the case for some algorithms in graph problems, that call decrease-key once per edge. Fast algorithms for computing minimum spanning trees and finding single-source shortest paths make essential use of Fibonacci heaps.

From a practical point of view, however, the constant factors and complexity of Fibonacci heaps make them less desirable than ordinary binary (or k-ary) heaps for most applications. As such, Fibonacci heaps are pre-dominantly of theoretical interest.

4.2 Structure of Fibonacci heaps

A Fibonacci heap is a collection of rooted trees that are min-heap ordered. Each tree obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. Each node contains a pointer to its parent and a pointer to any one of its children. The children are linked together in a circular, doubly linked list. In a circular, doubly linked list we can insert a node into any location or remove a node from anywhere in the list in $O(1)$ time. We can also concatenate two such lists in $O(1)$ time.

Each node also contains the number of children (**x.degree**) and a boolean-valued attribute x.mark, to indicate whether it has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked, and a node becomes unmarked whenever it is made the child of another node. Root nodes are unmarked.

We access a Fibonacci heap H by a pointer $H.min$ to the root of a tree containing the minimum key; we call this node the minimum node of the Fibonacci heap.

The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap. Finally, $H.n$ refers to the number of nodes currently in the heap.

In the following, the potential function of $\Phi(H) = t(H) + 2m(H)$ is used, where $t(H)$ is the number of trees and $m(H)$ is the number of marked nodes. The potential function captures the state of a system at a given point, and we can then calculate an amortized time as the actual time plus a constant times the **change** in state 'energy'. This provides a valid upper bound on running time.

4.3 Analysis: Running time of methods

4.3.1 Make-Heap

Creates a fibonacci heap with no trees. This gives a potential energy of $\Phi(H) = 0$, since $t(H) = 0$ and $m(H) = 0$. The total running time is thus the actual cost of $O(1)$.

4.3.2 Insert

Insert simply inserts a node as a root-tree in the Fibonacci heap, increasing the number of trees by one. To determine the amortized cost, let H be the input Fibonacci Heap and H' be the resulting Fibonacci heap. Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$. The increase in potential is thus: $((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1$.

Since the actual cost is $O(1)$, the amortized cost is $O(1) + 1 = O(1)$.

4.3.3 Find-Min

We store a pointer to the minimum node of a Fibonacci heap. As the potential is not changed (we alter no state in the tree), the amortized running time is the actual cost of $O(1)$.

4.3.4 Uniting two Fibonacci heaps

To unite two Fibonacci heaps, we concatenate the root lists of the two heaps H_1 and H_2 and then determine the new minimum node. The change in potential is:

$$\begin{aligned}
&\Phi(H) - (\Phi(H_1) + \Phi(H_2)) \\
&= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\
&= 0
\end{aligned}$$

Since $t(H) = t(H_1) + t(H_2)$ and $m(H) = m(H_1) + m(H_2)$. The amortized time is thus equal to the actual running time of $O(1)$.

4.4 ExtractMin

ExtractMin works by first making a root of each of the minimum nodes children and removing the minimum node from the root list. Then the root list is consolidated by linking roots of equal degree, until at most one root remains of each degree. To do this efficiently, an array is created that stores a current pointer to a root tree of a given degree.

We then consolidate the root list, to reduce the number of trees in the Fibonacci heap. The following steps are repeatedly executed until every root in the root list has a distinct degree:

1. Find two roots x and y in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$.
2. **Link** y to x: remove y from the root list, and make y a child of x. Then we increment the attribute $x.degree$ and clear the mark on y.

The array of degrees -> root pointers is used to look up whether there is another tree of the same degree as we have just increased x to, and if there is they link. If not, the entry in the array is updated to point to x.

The root list is then emptied and re-created based on the pointers in the array of degrees -> root pointers. Finally, we decrement $H.n$ by 1 and return a pointer to the deleted node.

There are several things we must estimate the cost of: extracting the minimum node, the main for-loop through the root list and within that the while-loop that links together trees.

The amortized cost of extracting the minimum node of an n-node Fibonacci heap is $O(D(n))$. Let H denote the Fibonacci heap prior to the ExtractMin operation. The actual cost of extracting the minimum node contributes $O(D(n))$ work from processing at most $D(n)$ chilren of the minimum node.

Upon calling consolidate, the size of the root list is at most $D(n) + t(H) - 1$: the original $t(H)$ root-list nodes, minus the extracted root node, plus the children of the extracted node, at most $D(n)$. Every iteration of the inner while-loop, we know that one root is linked to another, and so the total number of iterations of the while loop over all iterations of the for loop is at most the number of roots in the root list. Hence, the total amount of work performed in the for loop is at most proportional to $D(n) + t(H)$. The total actual work in extracting the minimum node is therefore $O(D(n)+t(H))$. The potential before extracting the minimum node is $t(H)+2m(H)$. The potential afterwards is at most $(D(n)+1)+2m(H)$, since at most $D(n)+1$ roots remain and no nodes get marked during this operation. The amortized cost is thus at most:

$$\begin{aligned}
&O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H)) \\
=&O(D(n)) + O(t(H)) - O(t(H)) \\
=&O(D(n))
\end{aligned}$$

if we scale up the units of the potential to dominate the constant hidden in $t(H)$. We will see later that $D(n) = O(lg\ n)$, giving an amortized running time of $O(lg\ n)$ for extracting the minimum node.

4.4.1 DecreaseKey

DecreaseKey on a node x (with parent y) works as follows: Decrease the key of x. If x.key $<$ y.key, cut x and turn it into a root-node. Then perform a cascading cut on y upwards, which will cut any node that is already marked, terminating when it reaches a root node or an unmarked node. Reaching an unmarked node will mark it. Finally, we check whether $x.key < H.min.key$, and if so update x to be the minimum node of H.

This process means that a node can lose one child (causing it to be marked), but losing another will cause it to be cut as it was already marked from losing a child before.

DecreaseKey takes $O(1)$ time, plus the time to perform the cascading cuts. Each recursive call of cascading cut also takes $O(1)$ time, giving $O(c)$ for c cascading cuts. The actual time of DecreaseKey, including recursive calls, is thus $O(c)$.

The call to cut x creates a new tree rooted at x and clears x's mark bit (which may have been false already). Each call of cascading cut, except for the last one, cuts a marked node and clears its marked bit. Afterwards, the Fibonacci heap contains $t(H) + c$ trees (the original $t(H)$ trees, $c - 1$ trees produced by cascading cuts and the tree rooted at x). It contains at most $m(H) - c + 2$ marked nodes ($c - 1$ nodes were unmarked by cascading cuts and the last call may have marked a node). The change in potential is therefore, at most:

$$((t(H) + c) + 2(m(H) - c + 2)) - (t(H) + 2m(H)) = 4 - c$$

This leads to an amortized cost of at most:

$$O(c) + 4 - c = O(1)$$

since, again, we can scale up the units of the potential to dominate the constant hidden in $O(c)$.

4.4.2 Delete

Deleting a node works by first setting its key to minus infinity, and then extracting the minimum. We already know that decreasing a key takes $O(1)$ amortized time, and extracting minimum takes $O(D(n))$ amortized time. As we will see that $D(n) = O(lg\ n)$, deleting a node runs in amortized time of $O(lg\ n)$

4.5 Bounding the maximum degree of a Fibonacci heap

Pages 523 - 526 of Lemma's and corollaries, need to be written and explained!

5 NP-Completeness: Disposition

1. P-Class, NP-Class, NPC-Class
2. Showing problems to be NP-Complete
3. Reductions and verifiability/certificates
4. P vs NP vs NPC

CLRS 34

6 NP-Completeness: Notes

NP-Complete problems are a class of problems, whose status is "unknown", in that no polynomial-time algorithm has yet been discovered for an NP-Complete problem, nor has anyone yet been able to prove that such an algorithm can not exist. Therefore, the so-called $P \neq NP$ problems, whose status is unknown. Several of the problems are tantalizing because they seem quite similar to problems that we **can** solve in polynomial time. For example, finding a single-source shortest path is solvable in polynomial time, finding a single-source longest path is **not**. Finding an Euler tour (cycle that traverses each each exactly once, but is allowed to visit vertices more than once) is solvable in polynomial time, a hamiltonian cycle (simple cycle that contains each vertex) is not.

6.1 P, NP, NPC Problems

The class P contains those problems that are solvable in polynomial time. More specifically, P class problems are solvable in $O(n^k)$ for some constant k, where n is the size of the input.

The class NP consists of those problems that are **verifiable** in polynomial time. That is, given a "certificate" of a solution, we can verify that the certificate is correct in polynomial time. For example, given a hamiltonian cycle $G = (V, E)$, a certificate could be a sequence of vertices $< v_1, v_2, v_3, \dots, v_n >$ of $|V|$ vertices. We can easily check in polynomial time that $(v_i, v_{i+1}) \in E$ for $i = 1, 2, 3, \dots, |V| - 1$ and that $(v_{|V|}, v_1) \in E$ as well.

Any problem in P is also in NP, since if a problem is in P then it is solvable in polynomial time. And so we can get the solution directly in polynomial time. $P \subseteq NP$. The open question is whether or not P is a proper subset of NP (meaning $P \subset NP$).

Informally, a problem is in the class NPC, and is called NP-Complete, if it is in NP and is at least as "hard" as any other problem in NP. If *any* NP-Complete problem can be solved in polynomial time, then *every* NP-Complete problem can be solved in polynomial time.

6.2 Showing problems to be NP-Complete

When we want to show a problem to be NP-Complete, we make a statement about how hard the problem is (or how hard we think it is), instead of how easy it is. We are trying to prove that no efficient algorithm likely exists. There are **three** key concepts in showing a problem to be NP-Complete:

6.2.1 Decision problems vs optimization problems

Many problems are **optimization problems**, where the solution has an associated legal value and we wish to find the best such value. For example, for the shortest path problem, we are given an undirected graph G and vertices u and v, and we wish to find a path from u to v that uses fewest edges. **However**, NP-Completeness applies to **decision problems** and not optimization problems, in which the answer is simply "yes" or "no".

However, it is often quite easy to convert an optimization problem into a decision problem, by imposing a bound on the value we wish to optimize. For example: Given a directed graph G, vertices u and v and an integer k, does a path exist from u to v consisting of at most k edges?

Decision problems are in a sense "easier" than optimization problems, since if we can solve the optimization problem then we can often trivially answer the decision problem. For example, if we find a path that solves the optimization problem of shortest path, it is very easy to solve the decision problem posed above by simply checking the length of the path. In other words, if the optimization problem is easy, then the decision problem is easy as well. Conversely, if the decision problem is hard, then the related optimization problem is hard as well.

6.2.2 Reductions

The above applies even when both problems are decision problems. We call the input to a particular problem an **instance**. For example, in PATH, an instance would be a particular graph G, particular vertices u and v of G, and a particular integer k.

If we have a decision problem A which we would like to solve in polynomial time, a decision problem B which we know we can solve in polynomial time and a transformation method that can transform input from the form that A accepts into the form that B accepts, then we can solve A in polynomial time as follows:

First transform the input into a form acceptable for problem B (polynomial time). Then run B and find an answer, in polynomial time. Then use that answer for A. Note that this assumes that when B answers 'Yes' then so does A, and when B answers 'No' then so does A. We can use this method to prove the opposite: That no polynomial-time algorithm can exist for problem B, if none exists for problem A.

Proof by contradiction. We wish to prove that no polynomial-time algorithm exists for a problem B. Suppose we have a decision problem A for which we already know no polynomial-time algorithm can exist. Suppose further that we have a polynomial-time reduction transforming instances of A to instances of B. Let us suppose that B has a polynomial time algorithm. Then, using the method covered above, we would have a way to solve problem A in polynomial time, thus contradicting our initial assumption that there is no polynomial- time algorithm for A.

For NP-Completeness, we cannot **know** that no polynomial-time solution exists for A, but we prove that problem B is NP-Complete on the assumption that problem A is also NP-Complete.

6.2.3 Encodings

In order to be able to converse in an encoding-independent fashion, we shall generally assume that problem instances are encoded in any reasonable, concise fashion, unless we specifically say otherwise. To be precise, we shall assume that the encoding of an integer is polynomially related to its binary representation, and that the encoding of a finite set is polynomially related to its encoding as a list of its elements, enclosed in braces and separated by commas.

Henceforth, we shall generally assume that all problem instances are binary strings encoded using the standard encoding, unless we explicitly specify the contrary. We shall also typically neglect the distinction between abstract and concrete problems.

TODO: Some actual proofs? Oh god NPC....

7 Randomized Algorithms: Disposition

1. Randomized Quicksort
2. Las Vegas, Monte Carlo
3. 1-sided vs. 2-sided
4. Bounding running times
5. Markov & Chebyshev's inequalities
6. Randomized Selection

Motwani & Raghavan Ch. 1.1, 1.2, Ch. 3.2, 3.3

8 Randomized Algorithms: Notes

8.1 Randomized quicksort

In randomized quicksort, we wish to sort a set of numbers S into ascending order. We first select a pivot y from S at random. Subsequently, each remaining element is compared to y and put into S_1 if smaller, S_2 otherwise. We recursively sort S_1 and S_2 in the same manner, and then output S_1 followed by y followed by S_2.

The running time of randomized quicksort is analyzed in terms of the number of comparisons made, as this dominates the running time of any (reasonably designed) algorithm. In particular, we wish to find the *expected* number of comparisons made.

For $1 \leq i \leq n$, let $S_{(i)}$ denote the element of rank i (the ith smallest element) in the set S. Thus, $S_{(1)}$ denotes the smallest element and $S_{(n)}$ the largest. Define the random variable X_{ij} to assume the value 1 if $S_{(i)}$ and $S_{(j)}$ are compared in an execution and the value 0 otherwise.

The total number of comparisons is then $\sum_{i=1}^{n} \sum_{j>i} X_{ij}$. The expected number of comparisons is then $\mathbb{E}[\sum_{i=1}^{n} \sum_{j>i} X_{ij}] = \sum_{i=1}^{n} \sum_{j>i} \mathbb{E}[X_{ij}]$ by Linearity of Expectation.

Let p_{ij} denote the probability that $S_{(i)}$ and $S_{(j)}$ are compared in an execution. X_{ij} can assume the values 0 and 1, so:

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

To facilitate determining p_{ij}, we view the execution of randomized quicksort as a binary search tree T, where each node is a distinct element of S. The root of the tree is the pivot y we choose initially; the left sub-tree contains elements in S_1 and the right sub-tree elements in S_2. The structure of the sub-trees is determined recursively by subsequent runs of randomized quicksort. Importantly, the root of the tree (y) is compared to elements in S_1 and S_2, but elements in S_1 are never compared to elements in S_2. Thus, there is a comparison between an $S_{(i)}$ and $S_{(j)}$ only if one is the ancestor of the other.

This makes T a random binary search tree. We are for this analysis interested in the level-order traversal of the nodes of T. Such a traversal is a permutation π obtained by visiting the nodes of T in increasing order of the level numbers, and in a left-to-right order within each level; recall that the ith level of the tree is the set of all nodes at distance exactly i from the root node.

To compute p_{ij} we make use of two observations:

1. There is a comparison between $S_{(i)}$ and $S_{(j)}$ if and only if $S_{(i)}$ or $S_{(j)}$ occur earlier in the permutation π than any element $S_{(l)}$ such that $i < l < j$. Intuitively, they are only compared if there is some pivot node l that puts i in S_1 and j in S_2. To see this, let $S_{(k)}$ be the earliest in π from among all elements of rank between i and j. If $k \notin \{i, j\}$ then $S_{(i)}$ will belong to the left sub-tree of $S_{(k)}$ while $S_{(j)}$ will belong to the right sub-tree of $S_{(k)}$, implying that there is no comparison between $S_{(i)}$ and $S_{(j)}$. Conversely, if $k \in \{i, j\}$ then it must be that there is a parent-child relationship between $S_{(i)}$ and $S_{(j)}$ implying that the two are compared.
2. Any of the elements $S_{(i)}, S_{(i+1)}, \ldots, S_{(j)}$ is equally likely to be chosen as the pivot and hence appear first in π. Thus, the probability that the first element is either $S_{(i)}$ or $S_{(j)}$ is exactly $2/(j - i + 1)$.

This establishes that $p_{ij} = 2/(j - i + 1)$. The expected number of comparisons is thus:

$$\begin{aligned}\sum_{i=1}^{n} \sum_{j>i} p_{ij} &= \sum_{i=1}^{n} \sum_{j>i} \frac{2}{j - i + 1} \\ &\leq \sum_{i=1}^{n} \sum_{k=1}^{n-i+1} \frac{2}{k} \\ &\leq 2 \sum_{i=1}^{n} \sum_{k=1}^{n} \frac{1}{k}\end{aligned}$$

It follows that the expected number of comparisons is bounded by $2nH_n$, where H_n is the *nth* Harmonic number, defined by $H_n = \sum_{k=1}^{n} 1/k$. We know that $H_n \sim ln\ n + \Theta(1)$, making the expected running time of randomized quicksort $O(n\ lg\ n)$

8.2 Las Vegas & Monte-Carlo

An algorithm that *always* gives the correct solution, but whose running time is variable or perhaps even unbounded, is a **Las Vegas** algorithm. In contrast, an algorithm that may sometimes produce an incorrect solution (although we can often bound the probability of this), but we know will always terminate, is a **Monte-Carlo** algorithm. One useful property of a Monte-Carlo algorithm is that we can reduce the risk of failure to be arbitrarily small by running the experiment repeatedly with independent random choices each time.

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

By definition, a Las Vegas algorithm is a Monte-Carlo algorithm with error probability 0. We say that a Las Vegas algorithm is an *efficient* Las Vegas algorithm if on any input its expected running time is bounded by a polynomial function of the input size. Similarly, we say that a Monte-Carlo algorithm is an *efficient* Monte-Carlo algorithm if on any input its worst-case running time is bounded by a polynomial function of the input size.

8.2.1 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 $1/t^2$. □

9 Hashing: Disposition

"Fast hashing" by Thorup. Sections 1, 2 and 3 up to and including 3.3. Proof in 3.3 only cursory, similar to proof in Section 2.

10 Hashing: Notes

11 Exact Exponential Algorithms & Fixed-parameter tractable problems: Disposition

1. O^*-notation
2. Parameterized Complexity
3. Travelling Salesman Problem using Dynamic Programming
4. CNF-Satisfiablility.

Formin & Kratsch, Ch. 1 Fixed-parameter tractable problems - Selected notes.

12 Exact Exponential Algorithms & Fixed-parameter tractable problems: Notes

12.1 O^*-notation

$\mathbb{O}^*$ notation is $\mathbb{O}$ notation where we suppress all polynomially bounded factors. For example $O(kn^k k^n) = O(n^k k^n)$ but we have $O^*(kn^k k^n) = O^*(k^n)$.

Different kinds of problems are measured based on different 'kinds' of input:

- Optimization problems on graphs are analyzed in terms of the number of vertices.
- Problems on sets are analyzed in terms of the number of elements.
- Problems on Boolean formulas are analyzed in terms of the number of variables.

There are often three broad categories of exact exponential algorithms:

- Subset problems. Maximum independent set.
- Permutation problems. Travelling Salesman Problem.
- Partition problems. Graph coloring.

12.2 Parameterized complexity

Parameterized complexity seeks to obtain algorithms whose running time can be bounded by a polynomial function of the input length, and, usually, an exponential function of a parameter. The parameter depends on the problem, e.g. the number of vertices in a graph, the number of variables in a formula, etc. However, it is unclear whether parameterized complexity is directly applicable, as one of the fundamental assumptions is that the parameter is independent of the input size.

12.3 Travelling Salesman Problem

Given cities $\{c_1, c_2, \ldots, c_n\}$ with distances $d(c_i, c_j)$, find the shortest path going through all cities exactly once and returning to the starting point. Different point of view: Find the permutation π of $\{1 \ldots n\}$ that minimizes $\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n), c_{\pi(1)}})$.

We use dynamic programming to compute for every pair (S, c_i) the optimal tour of S, starting at c_1 and ending at $c_i \rightarrow OPT[S, c_i]$. For $|S| > 1$, we have that $OPT[S, c_i] = min\{OPT[S \setminus \{c_i\}, c_j] + d(c_j, c_i) :$

$c_j \in S \setminus \{c_i\}\}$. That is, the optimal tour is the minimum distance from c_1 to c_j, plus the distance from c_j to c_i, for all subsets not containing c_i. For $|S| = k$, each step is $O(k^2)$, because we choose k possible c_i and for each find the preceeding best c_j. The total number of subsets is $\sum_{k=1}^{n-1} \binom{n}{k} k^2 = \mathbb{O}^*(2^n)$.

12.4 CNF etc...

FUUUUUDGE DET HER LAAAART!

13 Approximation Algorithms: Disposition

1. Performance ratio
2. Vertex-cover
3. Traveling-salesman
4. Set-covering
5. Randomization & Linear programming
6. Subset-sum problem

CLRS 35.1, 35.2, 35.3, 35.4, 35.5

14 Approximation Algorithms: Notes

An approximation algorithm is an algorithm that computes a near-optimal solution, often because the optimal solution is NP-complete and thus intractable or at least exponential in running time. One solution to that is to only work on small problem sets, where exponential time is fine. Another is to compute a near-optimal solution using an approximation algorithm.

14.1 Performance ratio

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

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

If an algorithm achieves an approximation ratio of $p(n)$ we call it a **$p(n)$-approximation algorithm**. Because we assume all solutions have a positive cost, these ratios are always well-defined. The approximation ratio of an approximation algorithm is never less than 1, since $C/C^* \leq 1$ implies $C^*/C \geq 1$. Therefore, a 1-approximation algorithm produces an optimal solution, and an approximation algorithm with a large approximation ratio may return a solution much 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.

14.2 Vertex-cover

A Vertex Cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v)$is an edge of G, then either $u \in V'$ or $v \in V'$ or both. The size of a vertex cover is the number of vertices in it.

The Vertex Cover Problem is to find a vertex cover of minimum size in a given undirected graph, this is an optimization version of an NP-Complete decision problem.

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 og 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. □

14.3 Travelling-salesman

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, and walks along this tree using a pre-order walk (parent first then child), and uses this as the solution.

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

Proof See page 1114.

14.4 Set-covering

14.5 Randomization & Linear programming

14.6 Subset-sum problem

15 Computational Geometry: Disposition

Computation Gemetry Ch. 9 (except 9.4 and 9.5). Delaunay Triangulation.

16 Computational Geometry: Notes