richardwu.ca

CS 466/666 Course Notes

Design and Analysis of Algorithms

Anna Lubiw • Fall 2018 • University of Waterloo

Last Revision: September 13, 2018

Table of Contents

Abstract

These notes are intended as a resource for myself; past, present, or future students of this course, and anyone interested in the material. The goal is to provide an end-to-end resource that covers all material discussed in the course displayed in an organized manner. These notes are my interpretation and transcription of the content covered in lectures. The instructor has not verified or confirmed the accuracy of these notes, and any discrepancies, misunderstandings, typos, etc. as these notes relate to course's content is not the responsibility of the instructor. If you spot any errors or would like to contribute, please contact me directly.

1 September 10, 2018

1.1 Overview

How to design algorithms Assume: greedy, divide-and-conquer, dynamic programming

New: randomization, approximation, online algorithms

For one's basic repertoire, assume knowledge of basic data structures, graph algorithms, string algorithms.

Analyzing algorithms Assume: big Oh, worst case asymptotic analysis

New: amortized analysis, probabilistic analysis, analysis of approximation factors

Lower Bounds Assume: NP-completeness

New: hardness of approximation

1.2 Travelling Salesman Problem (TSP)

Given graph (V, E) with weights on edges $W : E \to \mathbb{R}^{\geq 0}$ find a *TSP tour* (i.e. a cycle that visits every vertex exactly once and has minimum weight or $\min \sum_{e \in C} w(e)$).

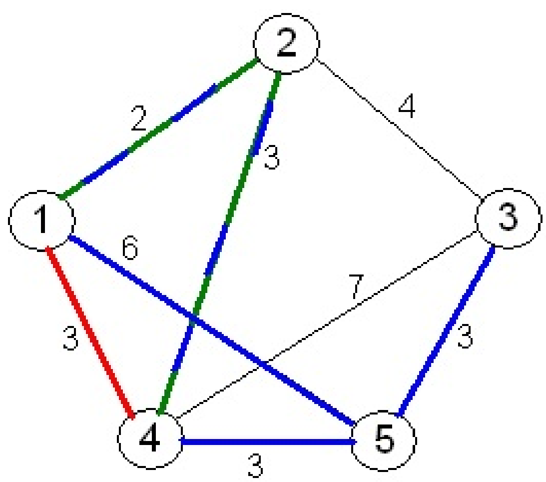


Figure 1.1: TSP tour is highlighted in blue.

Usually assume a **complete** graph (all possible $\binom{n}{2}$ edges exist). We can add missing edges with *high weight* to convert non-complete to complete.
Applications of TSP:

- School bus routes

- Delivery
- Tool path in manufacturing

To show the *decision version* of TSP (exist a tour of total weight $\leq k$) is NP-complete:

1. Show it is in NP (i.e. provide evidence (the tour itself) that there exists a TSP tour and show weights add up to $\leq k$)
2. Show a known NP-complete problem reduces in polynomial time ($\leq_p$) to TSP (the Hamiltonian cycle problem can be reduced to TSP)

1.3 Approach to NP-complete problems

For NP-complete problems we want to:

- Find exact solutions
- Find fast algorithms
- Solve hard problems

We can in effect only choose two: for hard problems we give up on either *fastness* (exponential time algorithms) or *exactness* (approximation algorithms).

1.4 Metric TSP

An appproximation exists for the **metric TSP** version, where:

- $w(u, v) = w(v, u)$
- $w(u, v) \leq w(u, x) + w(x, v) \quad \forall x$

An algorithm (1977) was proposed for metric TSP:

1. Find a minimum spanning tree (MST) of the graph
2. Find a tour by walking *around* the tree.

   Think of doubling edges of MST to get Eulerian graph (i.e. every vetex has even degree), which lets us find an Eulerian tour traversing every edge once.

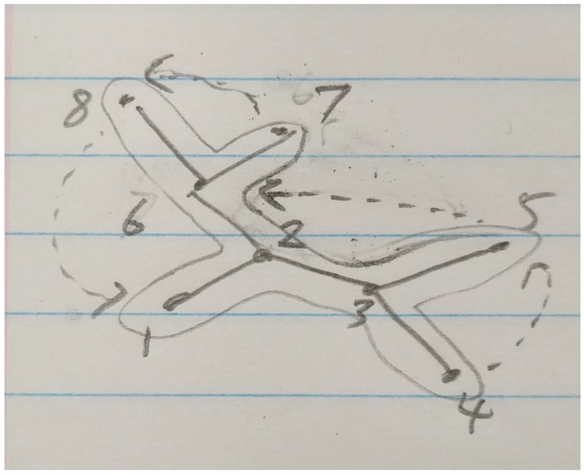


Figure 1.2: Eulerian tour is the solid line around the MST, the dotted lines show shortcuts taken, and the nodes are labelled in order.

3. Take shortcuts to avoid re-visiting vertices

   Instead of traversing a node twice (when walking around the MST), we take shortcuts and jump directly to the next unvisited node. By the triangle inequality our path should have a shorter path than if we actually traversed the MST edges twice, i.e.:

   $$l \leq 2l_{MST}$$

   where l is the length of our tour and l_{MST} is the length of the MST (remember we doubled every edge).

   Note the total path length we get will differ depending on which node we start with: thus one must attempt all paths to find the best.

   Lemma 1.1. This algorithm is a **2-approximation**, i.e.:

   $$l \leq 2l_{TSP}$$

   where l_{TSP} is the minimum length of TSP.

   Proof. We need to show $l_{MST} \leq l_{TSP}$.

   Take the minimum TSP tour. Throw out an edge. This is a spanning tree T. Since

   $$l_{MST} \leq l_T \leq l_{TSP}$$

   the result follows. □

   Exercise 1.1. Show factor 2 can happen.

Analyzing/implementing this algorithm (let n # of vertices, m # of edges):
Steps 2 and 3 take $O(n + m)$.
Step 1 is our bottleneck: we've seen *Kruskal's* (sorted edges with union find to detect cycles) and *Prim's* (add shortest edge to un-visited vertex) MST algorithms.
Prim's took $O(m \log n)$ using a heap. An improvement is using a *Fibonnaci heap* (1987) which improves runtime for MST to $O(m + n \log n)$. A further improvement uses a randomized linear time algorithm for finding the MST (1995).

Theorem 1.1. For general TSP (no triangle inequality) if there is a polynomial time algorithm k-approximation for any constant k, then $P = NP$.

Proof. Exercise (hint: start with $k = 2$ and the Hamiltonian cycle problem. Show the 2-approximation can be used to solve the HC problem). □

Can we improve factor of 2 for metric case? Yes (Christofides 1996):

1. Compute MST
2. Look at vertices of odd degree in MST (there will be an even number). Find a minimum weight *perfect matching* of these vertices.

   The MST and perfect matching is Eulerian: take an Eulerian tour and take shortcuts (as before).

Implementation: we need a matching algorithm - the best runtime (in this situation) is $O(n^{2.5}(\log n)^{1.5})$ (1991).

Lemma 1.2. We claim $l \leq 1.5 l_{TSP}$. Note that $l \leq l_{MST} + l_M$ (where l_M is the total length of the minimum weight perfect matching). We must show that $l_{MST} \leq l_{TSP}$ and $l_M \leq \frac{1}{2} l_{TSP}$.
Sketch: to show $l_M \leq \frac{1}{2} l_{TSP}$, we show the smallest matching is $\leq \frac{1}{2} l_{TSP}$.

Open question: do better than 1.5 for metric TSP. We know the lower bound is 1.0045 (if we could get 1.0045-approximation then $P = NP$).
There is also the **Euclidean TSP** version where $w(e) =$ Euclidean length. We can get ϵ-approximation $\forall \epsilon > 0$.

2 September 12, 2018

2.1 Data structures

Every algorithm needs data strutures. Assume knowledge of:

- Priority queue (heap)
- Dictionary (hashing, balanced binary search trees)

In this course, we look at fancier/better DSes and also amortized analysis.

2.2 Priority queue

Operations supported by a priority queue (PQ) are: insert, delete min (delete), decrease-key, build, merge.
We usually implement PQs with a **heap**: a binary tree of elements where the parent is $\leq$ than the left and right children (min-heap), therefore the min. is at the root.

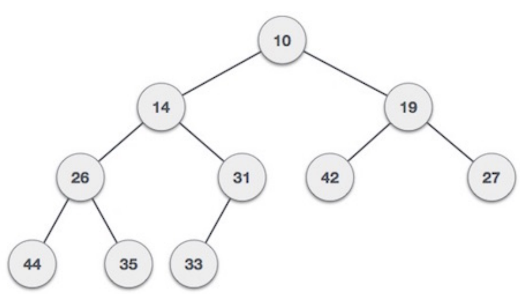


Figure 2.1: An example of a (min-)heap.

We assume that the shape is that of an almost perfect binary tree, where we always add a new element to the bottom right of the tree (if incomplete level) or the bottom left (start of a new level). We can store heaps in *level order in an array*, where for a given element indexed at i, acessing the parent via index $\lfloor \frac{i}{2} \rfloor$ and accessing the children via indexes $2i + 1$ and $2i + 2$.
The height of the tree is obviously $\theta(\log n)$. To implement each operation:

Insert Add new element in last position and bubble/sift up to recover ordering property. $\theta(\log n)$.

Delete min Remove root, it's the minimum. Move last position element to root and bubble/sift down (swap with smaller child). $\theta(\log n)$.

Derease-key Need only bubble/sift up (if $<$ parent). $\theta(\log n)$.

Build Repeated insertion is $\theta(n \log n)$.

Better approach: from bottom to top (after newly initialized heap in-place) bubble/sift down each element. $\theta(n)$.

2.3 Prim's algorithm

An application of heaps/PQs is for **Prim's MST algorithm**. Given graph with weights on edge, find spanning tree of minimum sum of edge weights.
For our given tree T so far, we find the minimum weight edge connecting to a new vertex. We begin with a PQ of all the edges and we will need to delete any newly added edge and any edges that lead to a newly added vertex.
Let $m = |E|$ number of edges and $n = |V|$ number of vertices. Every edge joins and leaves the heap once for a total of m times. We do delete min n times, so we have $\theta((m + n)\log m)$.
A better approach by using a heap of vertices and keeping track of shortest distance from T to vertex v gives us $\theta((m + n)\log n)$, which is only a constant time improvement since $\log m = \log n^2 = 2\log n$. We require the decrease-key operation here.
An even better approach for Prim's yields $\theta(m + n\log n)$ via *Fibonacci heaps*.

2.4 Binomial heaps

Binomial heaps improve the merge operation for heaps (which we can use for all other operations).
We use pointers to implement trees and each parent has an arbitrary number of k children (not necessarily binary tree) while maintaining heap order where parent is $\leq$ key of all children. We thus need to relax the shape; we also allow *multiple trees* for a given heap.
We define binomial trees B_k in terms of their rank k, which also coincides with the degree of the root.

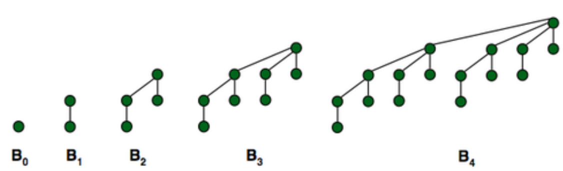


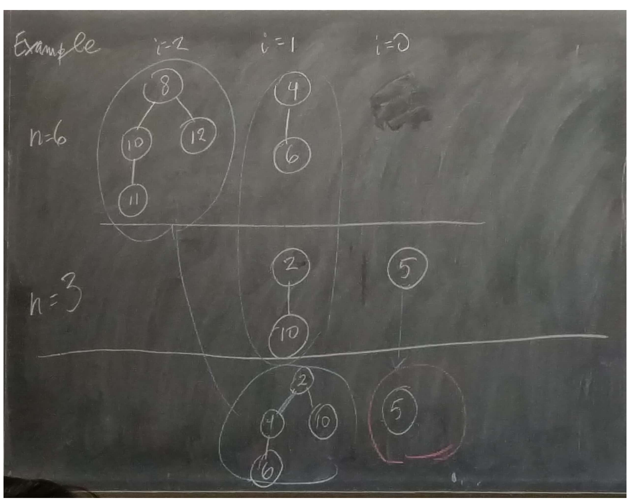
Figure 2.2: Example of five binomial trees with ranks $0, 2, \ldots, 4$.

In general, the number of node in B_k is $2^k = 2^{k-1} + 2^{k-1}$.
The height of B_k is k since, by induction, we have the recurrence $height(k) = 1 + height(k-1)$.

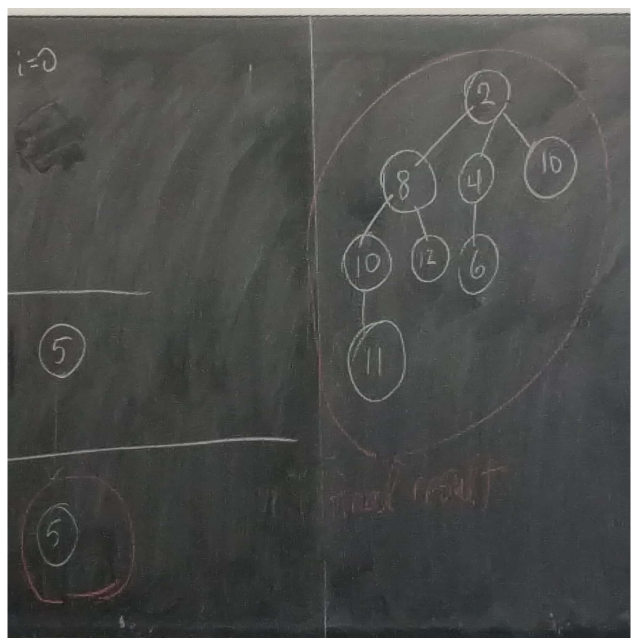
The number of nodes at depth i in B_k is

$$\binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$$

Brief proof: given k items from which we want to choose i items, we can either picked the first item or not: if we did not pick the first element, then we need to pick i items from the remaining $k-1$ items; if we did pick the first element, we need to pick $i-1$ items from the remaining $k-1$ items.
Note that B_k's only permit powers of 2 number of elements: thus a **binomial heap** for n elements use a collect of B_is (heap ordered), at most one for each rank.
E.g. for $n = 13$, we have $13 = 2^3 + 2^2 + 2^0$ (from binary 1101 so we use B_3, B_2, B_0.
It *does not matter* which B_i's contain a particular element.
We use $\theta(\log n)$ trees for a given binomial heap with n elements (from binary number expansion).
For merging two binomial heaps, we follow addition of two binary numbers (which are our bitmasks of trees). E.g for adding a heap with 6 elements to a heap with 3 elements, we add $110 + 11 = 1001$ resulting in a binomial heap with a B_3 and B_0 tree.
When merging two trees of the same rank k, we simply take the tree with the larger root and add it as a children of the root of the other tree.



(a) We carry through the B_0 from the second heap and merge the B_1 trees from two binomial heaps by making the tree with the larger root the children of the other root.



(b) After merging the newly B_2 with the B_2 tree from the first heap into a B_3 tree, we get our final resulting binomial heap with a B_0 and B_3 tree.

Analysis of operations:

Merge Joining two B_i's take $\theta(1)$. We join up to $\log n$ trees so we have $\theta(\log n)$.

Insert Merge binomial heap with single B_0 (one new element): again $\theta(\log n)$ (since we have the worst case when we insert into a heap with $2^m - 1$ elements.

Delete min Takes $\theta(\log n)$ to find the minimum by checking roots of all trees. Once removing said tree from B_k, we end up with $k-1$ trees $(B_{k-1}, \ldots, B_0)$ which we need to merge with the other trees (merge operation is also $\theta(\log n)$) so we have $\theta(\log n)$ overall.

Decrease-key After decreasing key, we bubble/sift up as necessary like before, which takes $\theta(\log n)$ since each tree has height at most $\log n$.

Build Repeated insertion appears to be $O(n \log n)$, but in fact it is $\theta(n)$. This can be seen by repeated addition of 1 in binary to our cumulative total: we only do merges at certain times.