CSCI270 Week 4

Jacob Ma

September 28, 2022

1 Review: Exchange Argument

- If it gets no worse, use induction to say the optimal algorithm is the current algorithm.
- If it gets worse, there exist a contradiction, thus no need to conduct further induction.

2 Minimum Spanning Tree

Problem 1

Given an undirected connected graph $G = (V, E)$ with edge weights / costs $w(e) \geqslant 0$.
Assume for simplicity that all edge wights are distinct.
Goal: find a connected sub graph (set E' of edges that is a subset of E) such that (V, E') is still connected, and E' has minim total cost (sum of the $w(e)$ for e in E') among all such sets.

Applications:

(1) Connect the vertices V at minim cost (e.g., road network, computer network, rail network, $\cdots$) to ensure that everyone can still get everywhere.

Note: this may result in unnecessarily long paths, and would be not fault-tolerant at all (removing an edge might disconnect large parts).

$\Longrightarrow$ Real solutions would not just minimize cost, but build in redundancy and shorter paths.

However, solving this problem will still be a central part of finding better solutions (with more realistic objectives).

(2) As a subroutine for solving other problems, in particular Traveling Salesman.

If you had a cycle, you could take out any one edge and make the solution cheaper $\Longrightarrow$ optimum solution is acrylic.
It must also connect all of the vertices, so we want an acrylic connect edge set of minimum cost.
$\Longrightarrow$ Minimum Spanning Trees(MST) ["Spanning" refers to spanning or connecting all of the vertices]

Problem 2

What do we know about optimum solution? Which edges will it definitely include, or definitely not include?

Definition 2.1: Cut of a graph

A "cut" of a graph $G = (V, E)$ is a partition of the nodes into two sets $S, \overline{S} = V \backslash S$
When typing, we write the complement as $\overline{S} = V \backslash S$, so the cut is $\left(S, \overline{S}\right)$. An edge $e = (u, v)$ "crosses" the cut $\left(S, \overline{S}\right)$ of one of its endpoint's is in S and the other is not in S (so in $\overline{S}$).

Theorem 2.2: Cut Property

If the edge e is cheapest among edges crossing some cut $\left(S, \overline{S}\right)$, then e is in every Minimum Spanning Tree.

Proof. We prove by contrapositive.
Let T be any (spanning?) tree not including e, and e cheapest across the cut $\left(S, \overline{S}\right)$. We will show that T is not a MST.
Adding e to T creates a cycle C.
Our goal is to show that C contains another edge e' that is more expensive than e.
Then, $T + \{e\} \setminus \{e'\}$ is a cheaper solution.
So T cannot be a MST.
To show that e' exists, remember that e crossed the cut (S, S'), so u is in S and v is in S'. $C \setminus \{e\}$ is a path from u to v which starts in S and ends in S', so it must cross from S to S' at least once.
So $C \setminus \{e\}$ contains another edge e' from S to S'.
$w(e') > w(e)$ because e was cheapest across the cut. So $T \setminus \{e'\} + \{e\}$ is cheaper than T, so T is not a MST. □

Based on the Cut property, we get some kind of generic algorithm:

(1) Start with no edges selected.

(2) While the selected edges don't connect the entire graph, add an edge which is known to be cheapest across some cut.

Instantiation 1: Kruskal's Algorithm

(1) Sort edges by increasing weight $w(e) \implies \theta(m \log m)$

(2) In this order, we go through the edges. $\implies \theta(m)$

- When looking at edge e, if it creates a cycle with the edges already selected, discard it; otherwise, pick it.

Proof. Correctness Proof:
Kruskal produces connected components, and each edge that is added merges two components. When e is added, merging C_1, C_2, it is cheapest across the cut $\left(C_1, \overline{C_1}\right)$, and also across $\left(C_2, \overline{C_2}\right)$.
So e is cheapest across some cut, so it is in the MST.
So the output of Crustal is a subset of the MST.
If the output contained fewer than $n - 1$ edges, it would not be connected, so there is a partition $\left(S, \overline{S}\right)$ with no

edges selected.
Because the input graph was connected, there must have been at least one edge connecting S to $\overline{S}$. Kruskal would included the first such edge.
So the output contains $n-1$ edges, so it must equal to MST. □

Faster tie using efficient Union-Find data structures: can implement lookup and updates in time $O(n)$.

Definition 2.3: Efficient implementation of Union-Find Data Structure

- for each node, we have a pointer to a parent.
- these pointers define a forest.
- the root of the tree of a node will gibe the identity of the component it belongs to (can find it by followings pointers until reaching a root)
- by being careful with the rule for merging, can ensure ?????

With optimizations, this improves to $O(\log^* n)$ amortized.

Resulting running time: $O(m \log m)$ [sorting] $+ O(m \log^* n) = O(m \log m) \implies$ Sorting is now the bottleneck.
Instantiation 2: Prim's Algorithm

- Start with $S = \{s\}$ (s is an arbitrary start index)
- Until $S = V$ (all nodes in the graph), in each iteration:
  - Find the cheapest edge $e = (u, v)$ between S and $\overline{S}$.
  - Add e to T, and add v to S.

Proof. Correctness: Whenever an edge e is added, it is explicitly chosen as cheapest between S and $\overline{S}$, so each added edge is cheapest across some cut.
The algorithm adds $n-1$ edges, so the output must be the MST. (Connectivity is implicitly used to show that an edge can always be found and / or that the algorithm terminates.)
Total runtime: $O(mn)$, where m is number of edges, n is number of vertices. □

Less good version: use a min heap, containing all edges crossing the cut $\left(S, \overline{S}\right)$ at the current iteration.
Add edges when a node gets added.
The min-heap will always contain edges crossing the cut, as well as some leftover edges inside S.
Find the minimum (at the root) in each iteration.
In any iteration, we add degree(v) edges to the heap (when v is added to S), each taking times $O(\log m)$.
So the total is $O(m)$ times the sum of degrees, which is $O(m \log m)$
$\implies$ Running time is $O(m \log mm) = O(m \log n)$. (because $m \leqslant n^2$, $\log m \leqslant \log\left(n^2\right) = 2 \log n = O(\log n)$)
Using Fibonacci Heaps, this improves to $O(m + n \log n)$.
Better approach: Use a min heap, containing all nodes that are not in S. For each node, keep the minimum cost of any edges connecting it to S. Find the minimum-cost node to add next. Based on the edges from this node to the nodes in the heap, possibly update their cost to a smaller value.

Each edge leads to at most one update of a value in the heap $\implies$ running time is $O(m \log n)$. (This is exactly how you would implement Dijkstra.)

3 Divide and Conquer

Definition 3.1: Divide and Conquer

High level idea of Divide & Conquer:

- Take a problem instance l of size n
- Divide it into smaller instances $l(1), l(2), \cdots, l(k)$
- Solve each of the $l(j)$ separately , resulting in $Sol(j)$
- Do some post processing work to produce a solution Sol from $Sol(j)$.

Most frequently, $k = 2$.
Often (but not always), the sub problems $l(j)$ have the same size, and are disjoint parts of the input, of size $\frac{n}{k}$.

Example 3.2: Merge Sort $(a[], L, R)$.

- if $R = L$, then nothing to do
- otherwise, let $m = \frac{R+L}{2}$, rounded down
- Merge Sort $(a[], L, m)$;
- Merge Sort $(a[], m+1, R)$;
- Merge $(a[], L, m, R)$

Example 3.3: Merge $(a[], L, m, R)$.

- $i = L; j = m + 1; k = 0;$
- b = a new array of size $R - L + 1$ [0 indexed]
- while $(i \leqslant m) \,||\, j \leqslant R$
  - if $(j > R \,||\, (i \leqslant m) \&\& a[i] \leqslant a[j]) \{[k] = a[i]; i++; k++;\}$
  - else $\{b[k] = a[j]; j++; k++\}$
- for $(k = 0; k \leqslant R - L; k++) \{a[L+k] = b(k);\}$

Goal: Prove that Merge Sort is correct.