Algorithms and Datastructures (HS2024) Week 11

Rui Zhang

December 1, 2024

Contents

1 Revision Theory

1.1 Why is Dijkstra not enough?

Last week, we introduced Dijkstra algorithm. This algorithm has a very important caveat though: It only works for a graph with only positive edge costs.

If we have negative edge costs, Dijkstra may fail and calculate completely wrong minimum distances. As an exercise, find a graph where this happens. The reason why Dijkstra fails in some graphs with negative edge weights is because Dijkstra is what we call a "greedy" algorithm. We sorted the vertices that we explore by their current distances from the starting vertex s and removed them from the priority queue, only to mark them as visited and therefore prevent any future visiting and thus updating of their minimum distances from s. This did help us achieve a linear runtime, however, this is what ultimately leads to Dijkstra's demise.

1.2 Bellman-Ford

Now, we consider an algorithm that removes this restriction. Meet the Bellman-Ford-Algorithm which can find shortest paths to every vertex starting at some vertex s and even detect negative cycles in $O(n \cdot m)$

One way you can view Bellman-Ford is "Dijkstra but less greedy". We also start off with an array of minimum distances from a starting vertex s, called d (then, $d[v] :=$ minimum distance of vertex v from s). In Dijkstra however, we only considered the edges incident to the vertices we have explored. Using these edges, we would update the minimum distances of the vertices. Now, we will go through all edges in each iteration of the Bellman-Ford algorithm and update the minimum distances of the vertices, just like in Dijkstra via the formula

$$d[v] = \min_{(u,v)\in E} d[u] + c(u,v)$$

In the exercise class, we saw an example of this in action. And, you have already seen one in the lecture as well. What you might have noticed is that for each iteration, we are basically using another edge. Thus we conclude that in the first iteration, we calculate shortest distances for paths of length 1 (using 1 edge), in the second, we calculate shortest distances for paths of length 2 (using 2 edges) and so on.

Thus, we only need to do this procedure we just mentioned $n-1$ times. This would result in a total runtime of $O(n \cdot m)$. This is because iff we did an n-th iteration, we would be considering shortest paths of n edges, but that would be $n+1$ vertices, which would imply a cycle somewhere, as we only have n vertices. Since we are computing shortest paths, cycles cannot be helpful, computing the n-th iteration would not help us much then,....

Unless...

1.2.1 Detecting Negative Cycles

Pause. In science, always challenge your own assumptions. Trust nobody. Not even yourself. Who said cycles cannot be helpful? What if that cycle if is a cycle with a negative total cost? Then computing the $n - th$ iteration would indeed lead to better distances. Pause again. Negative cycles are the achilles' heel of our current problem of finding shortest distances! because if a negative cycle exists, we can just repeat that cycle and therefore we would not actually have a shortest path.

Hey look! Now we have found a way to detect negative cycles. We just compute the n-th iteration of Bellman-Ford and check if any distance was updated / improved. If yes, then we have a negative cycle.

1.3 Minimum Spanning Trees (MST)

Given a directed, weighted **and connected** graph $G = (V, E, c)$, with non-negative edge-costs $c : E \to \mathbb{R}^{\geq 0}$ we now want to find a subset of edges $A \subseteq E$ with minimum total summed up cost such that the induced graph $G' = (V, A)$

is still connected. Note that such a graph is always a tree. Otherwise, we would have a cycle and we could remove any edge from that cycle to receive a subset A of less total cost.

Our intuition might tell us that the edges we want to include in this tree are the edges of minimum cost. That intuition is correct. Indeed, **safe edges** are edges that are present in every MST and the cut-principle confirms that edges of minimum cost are the safe edges. From this, we develop our first algorithm:

1.3.1 Boruvka's Algorithm

Boruvka's Algorithm starts with just the vertices from the graph (no edges). Note that if we have n vertices in the original graph, we would have n connected components now. We will construct an MST by adding edges that connect different connected component. By the cut principle, these edges will have to be the edges of least cost connecting different connected components. After each iteration, our connected components will grow in size, and the number of connected components will be reduced. And after a certain amount of iterations, we will have only one connected component (our MST) left.

The runtime of this algorithm can be analyzed as follows: in each iteartion, to find the connected components we may apply DFS. Then, we would go through all edges and check if they connect different connected components. For each connected component, we would find a minimal incident edge. This will take $O(n + m)$. Now we only need to know the number of iterations.

in each iteration, the worst case is if every two connected components share the same minimal edge. In this case, we would at least halve the number of connected components each iteration. Since we start off with n connected components, we will need less than $\log(n)$ many iterations, with a total runtime of $O(n \log(n))$.

1.3.2 Prim's Algorithm

In Boruvka, we started off with all vertices at once. But using the cut-principle, we may also just start off with one vertex and expand the MST from there. That is precisely the idea behind Prim's algorithm.

Prim's algorithm starts off at some vertex s and marks it as "visited" / "added to the MST". In each iteration of Prim's algorithm, we find all edges incident to the "visited" vertices and take only those that connect to non-visited vertices too. We find the minimum of all these edges (cut-principle) and adds this edge and its succeeding vertex to the MST and its set of "visited" vertices. Once all vertices are "explored", we may return the edges we have collected as the MST.

Now, let us take a look at the pseudocode:

```
Prim(G, s): // s is the starting vertex, G = (V, E, c)
  P = PriorityQueue(V) // all vertices get a priority of infinity
  decreaseKey(P, s, 0)
```

```
while !isEmpty(P):
  u = extractMin(P)
  mark u as visited
  for (u, v) in E, v not visited:
    decreaseKey(P, v, min{current key of v in P, c(u,v)})
```

As you will notice, this is literally just Dijkstra, but instead of visiting the vertices / edges in order of their distances to the starting vertex s, we order them by their edge costs via cut principle.

Same as with Dijkstra, we have a runtime of $O((n + m) \cdot \log(n))$.

2 Exercise Sheet

2.1 Exercise Sheet 11 - Priority List

1. - 11.4 - in past exams, this type of exercise has always been very relevant
2. - 11.3 - although not bonus, this exercise has often been exam relevant. It communicates the idea of what I like to call "layered graph modelling"
3. - 11.1, 11.2 - you would not want to miss out on bonus points
4. - 11.5 - I wish to exclaim my disdain towards the Deutsche Bahn.

2.2 Exercise Sheet 9 - Feedback

- When writing the dimensions of a DP table, please do not only specify that it has n entries, or that it has one dimension. Write something like $1 \times n$
- As mentioned in a moodle message from the Head-TAs, you should answer the "calculation order" questions in DP via a pseudocode of nested loops
- Many of you had trouble with exercise 9.4. Most of you defined a table entry just like the hint specified, but your recursion was something like $DP[i] = max_{(v_i,v_j)\in E} DP[j] + 1$ however, this would mean that v_j is a successor of v_i. For one, this recursion would not make any sense then. Additionally, since often this recursion was combined with a calculation order consistent with topological ordering, $DP[j]$ was not even calculated at the point of calculating $DP[i]$.

  The actual solution was to create a reversed Adjacency List.

3 Supplementary Exercises

3.1 Why Dijkstra is not enough

Find a example of a graph where Dijkstra fails to calculate minimum distances correctly. You will need to assign negative edge weights / costs