Algorithms and Datastructures (HS2024)
Week 10

Rui Zhang

November 23, 2024

Contents

1 Revision Theory

Last week, we looked at exploring and traversing graphs via the DFS algorithm as well as topological sortings. This week, we will continue with the topic of traversing graphs. However, this time we will be asking ourselves the question how to find shortest paths between vertices.

1.1 Breadth First Search

BFS is the counterpart to DFS. Here, intuitively, instead of exploring the graph by going "in depth", we traverse "layer by layer" and are therefore able to find the shortest path to other vertices starting at some initial vertex s. BFS and DFS are algorithms you should be able to implement, so here is the pseudocode:

```
BFS(s): // we are starting at some initial vertex s.
  Q = {s} // this is supposed to be a queue datastructure
  while Q is not empty:
    u = dequeue(Q)
    mark u
    for v a successor of u, v not marked yet:
      enqueue(Q, v);
```

This code is very simple. We start off with a queue initially containing the starting vertex s. From there on out, we also have a for loop going through all of the successors of the current vertex u we are traversing. **Instead of immediately traversing these successors, we add them to the queue and continue with the next vertex in the queue.** Now, if you imagine a BFS algorithm on some graph, then you will realize that the structure of a queue (FIFO-principle, first-in-first-out) forces the algorithm to traverse the graph in a layer-by-layer order.

From this layer-by-layer order, we can find the shortest-paths from the starting vertex s to every other vertex $v \neq s$ in the graph.

1.1.1 Shortest Path Trees for Unweighted Graphs

In a directed graph, you can also visualize the "shortest path tree" starting from some vertex $s \in V$ via the BFS algorithm. For every one of the vertices we add to the queue in the pseudocode, all we have to do is highlight / draw the edge from the predecessor to the vertex we just added.

We will look at an example in class.

1.1.2 Enter & Leave Numbers / Order

Similar to the pre- and post- numbers in DFS, we have enter and leave numbers for BFS. However, the structure is much less exciting. If we modify our pseudocode to be like the following:

```
BFS(s): // we are starting at some initial vertex s.
  Q = {s} // this is supposed to be a queue datastructure
  while Q is not empty:
    u = dequeue(Q)
    leave[u] = T
    T = T + 1
    mark u
    for v a successor of u, v not marked yet:
      enqueue(Q, v);
      enter[u] = T
      T = T + 1
```

These numbers represent when a vertex enters and leaves the queue. Due to the FIFO structure of a queue, we have that the enter-order is the same as the leave-order.

1.1.3 Runtime

It's the same as DFS: $O(m+n)$. The same explanation and reasoning applies as well. As an exercise, you may write the proof yourself. I cannot be bothered to ctrl+c ctrl+v rn.

1.2 Weighted Graphs

Before we approach Dijkstra, we have to establish the concept of weighted graphs, which will be of heavy relevance for future topics as well. Up until now, we have had directed graphs $G = (V, E)$ as a tuple. We will now introduce another variable in this tuple: $G = (V, E, c)$ and now we have a directed, weighted graph. This is because c is a function $c : E \to \mathbb{R}$ which assigns each directed edge a weight / cost of traversing this edge. Think of energy consumption when going for a run, fuel consumption when travelling by car or the cost for tickets when travelling in a foreign country.

Now, we want to consider not only the cost of edges, but the cost of an entire walk. For this purpose, we define for a walk $W = (v_0, v_1, v_2, \dots, v_l)$ of length l the cost of that walk $c(W) = \sum_{i=0}^{l-1} c(v_i, v_{i+1})$ to be the sum of all edge-costs along that walk.

1.3 Dijkstra's Algorithm

Now that we have weighted graphs, we would want to find shortest-paths as in these sort of graphs as well. However, note that such a shortest path may not always exist. Because what if we have a so called ”negative cycle”? So before trying to find shortest-paths in a weighted graph, always make sure that there is no negative cycle.

Before we start with the actual algorithm, let us consider the following piece of intuition:

”When traversing a weighted graph, if we always only explore vertices ordered by their distances from the starting vertex s, then we should (probably) get the shortest path from s to every other vertex in the graph”

In other words, the idea similar to BFS, but instead of a Queue, we have a **Priority Queue:**

```
Dijkstra(s): // we start off at s
  P = PriorityQueue(V)
```

```
// initialize the PriorityQueue, which is a Min-Heap
// where every vertex has a key of infinity
// Note that the keys represent the shortest distances
// of each vertex to the starting vertex s we have
// found up until now. Infinity means that
// we have not traversed this vertex yet.

mark s as visited
decreaseKey(P, s, 0)
// we are setting the key of s in the PriorityQueue to be 0.
// Of course, this makes sense.
// which means that s is now the root of the min-heap

while P not empty:
  u = P.dequeue()
  // the get vertex that is at the front of the priority
  // queue is dequeued. This is equivalent to "extractMin"
  // in a heap, so we are getting the vertex with the least
  // distance from s we know up until now

  mark u as visited
  for (u, v) in E, v not visited / marked yet:
    decreaseKey(P, v, min{current key of v,
                          current key of u + c(u, v)})
```

1.3.1 Shortest Path Trees for Weighted Graphs

In a directed graph, you can also visualize the "shortest path tree" starting from some vertex $s \in V$ via the BFS algorithm. For every vertex v we dequeue from the PriorityQueue we will add the edge from the predecessor p from which this vertex got its key and draw / highlight this edge (p, v). We will also look at an example in class.

1.3.2 Runtime

The runtime of this algorithm depends heavily on how we implement the PriorityQueue. We may choose to just use an array like you did in class. However, this will result in quadratic runtime.

Instead, remember the runtime for Min-Heaps we learnt a few weeks ago. We will use Min-Heaps to implement this PriorityQueue and get that each operation "decreaseKey", "dequeue" is in $O(\log(n))$ and that the initialization is in $O(n \log(n))$. We traverse every vertex only once and for each vertex, we consider their successors to potentially update their shortest distances in the

PriortyQueue. This results in a total runtime of:

$$\begin{aligned}
&O(n\log(n)) + \sum_{v\in V} O(1+\deg_{out}(v)) * O(\log(n)) \\
&= O(n\log(n)) + O(\log(n)) \cdot \sum_{v\in V} O(1+\deg_{out}(v)) \\
&= O(n\log(n)) + O(\log(n)) \cdot O(n+m) \\
&= O(n\log(n)) + O((n+m)\cdot \log(n)) \\
&= O((n+m)\cdot \log(n))
\end{aligned}$$

2 Exercise Sheet

2.1 Exercise Sheet 10 - Priority List

1. - 10.2), 10.4) - Bonus and exam-like
2. - 10.5), 10.1) - Deeper understanding of Dijkstra and graphs
3. - 10.3) - Graph Modelling, will still come in handy in A&P though.

2.2 Exercise Sheet 8 - Feedback

- Try to be more mathematically precise when writing proofs for graphs. Most of your submissions were done quite well in terms of intuition and logic, however, showing that you can write the proof down also in mathematical notation will often deepen your understanding of the proof itself.
- Make sure to revise the terminology for graphs. This will help in avoiding long / shady explanations of concepts. For example, in an exam, you can just write "connected components" then, instead of something like "part of a graph".

3 Supplementary Exercises

3.1 Breadth First Search to find shortest distances

Modify the pseudocode from the BFS section such that at the end, we have an array D of size $n := |V|$ and $D[v] :=$ the shortest distance from the starting vertex s to vertex v. You may assume we have a connected graph.

3.2 Breadth First Search to find shortest paths

Modify the pseudocode from the BFS section and write the pseudocode for a function $shortestPath(v)$, such that $shortestPath(v)$ returns in a linked list the shortest path from s to v.

3.3 DFS, Iteratively

In class, you were also told that DFS can be implemented iteratively just like BFS. Write the pseudocode for an iterative DFS.

3.4 Dijkstra for finding shortest paths

Modify the pseudocode from the Dijkstra section and write the pseudocode for a function $shortestPath(v)$, such that $shortestPath(v)$ returns in a linked list the shortest path from s to v.

Hint: Use a "predecessors array".