# Recitation 10

Shortest Paths

10.1   Announcements

- *ShortLab* has been released, and is due **Thursday night**. It's worth 125 points.

- *SegmentLab* will be released on **Thursday**.

10.2   Dijkstra's Algorithm

For this section, we'll be using Dijkstra's algorithm as implemented in the textbook.

Task 10.1. *Run Dijkstra's algorithm on the following, starting from* GHC. *Trace the process by writing the most recently popped vertex and the contents of the priority queue immediately before each* deleteMin *operation. Also write the returned mapping of each vertex to its shortest path distance.*
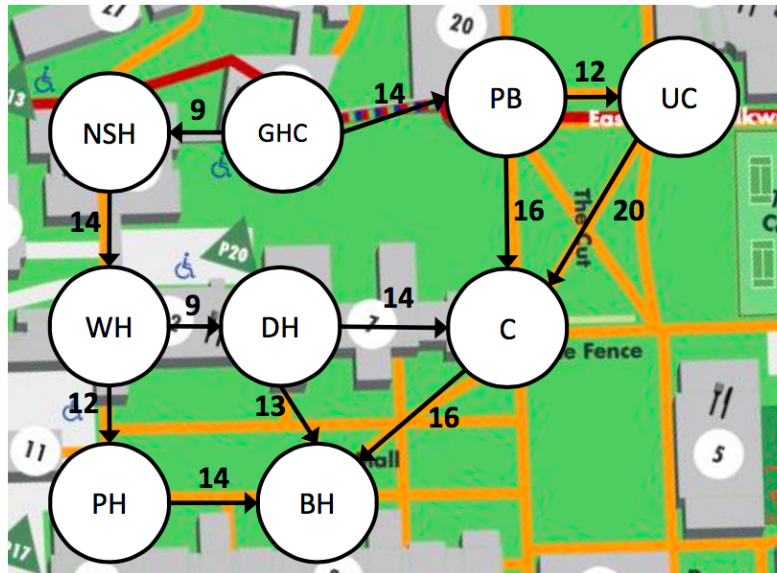
Task 10.2. *Consider the following simple modification to Dijkstra's algorithm:*

When relaxing the neighbors of a vertex, we only insert a neighbor into the priority queue if it has not already been visited.

Does this modification improve the worst-case cost bounds of Dijkstra's algorithm?

Task 10.3. *Priority queues sometimes support a* `decreaseKey` *operation which attempts to decrease the priority associated with some value. For example, in the priority queue* $\{(15, A), (42, B)\}$, *we could apply* `decreaseKey` *on* B *with a new priority of* 11 *to get back the priority queue* $\{(11, B), (15, A)\}$. *If the newly specified priority is greater than the original, then this operation simply does nothing.*

Suppose that you are given a priority queue which supports `insert` *and* `decreaseKey` *in constant time, and* `deleteMin` *in* $O(\log |Q|)$ *time.[a] Describe how to use this data structure to improve the asymptotic performance of Dijkstra's algorithm. You may assume the graph is enumerated.*

---

[a]These bounds are supported by implementations such as Fibonacci and Brodal heaps.

10.3 A*

Dijkstra's algorithm computes shortest paths between the source and every other vertex in the graph. But what if we only care about the shortest path to a specific "target" vertex?

In the example graph given above, if you are trying to get from GHC to BH, you intuitively know that the shortest path probably doesn't pass through the UC. As you work your way towards the goal, you are guided by an internal *heuristic* which helps you estimate how far each building is from BH and choose the most promising option.

We can apply a similar approach to Dijkstra's algorithm. At each step, we'd like to visit the "most promising" candidate among the vertices in the frontier. To determine how promising a candidate is, we need an *estimate* of how far that candidate is from the destination. Specifically, we'll assume we have a heuristic function $h : \mathbb{V} \to \mathbb{R}^+$ which maps vertices v to an estimate of the length of the shortest path between v and the destination.

The A* algorithm is identical to Dijkstra's except that it orders its priority queue by $d(v) + h(v)$ rather than by $d(v)$, and terminates as soon as it finds the target. We also need to explicitly store the distance to each vertex in the priority queue. Here is the code:

Algorithm 10.4. *A* search with a consistent heuristic*

```
1  A* h (G,s,t) =
2    let
3      search (X,Q) =
4        case PQ.deleteMin Q of
5          (NONE, _) ⇒ ⊥
6        | (SOME (_,(v,d)), Q') ⇒
7            if v = t then d else
8            if v ∈ X then search (X,Q') else
9            let
10               X' = X ∪ {v ↦ d}
11               relax (Q,(u,w)) = PQ.insert (Q,(d+w+h(u),(u,d+w)))
12               Q'' = iterate relax Q' (N⁺_G(v))
13           in
14               search (X',Q'')
15           end
16    in
17      search ({}, PQ.singleton (h(s),(s,0)))
18    end
```

In order to ensure correctness, we need h to be *consistent*, meaning that for every two vertices u and v, $h(u) \le \delta(u,v) + h(v)$ where $\delta(u,v)$ is the shortest path from u to v. Furthermore, the heuristic value of the destination must be 0: $h(t) = 0$. As an exercise, try proving that A* always finds an optimal path when using a consistent heuristic.

It turns out that all *consistent* heuristics are also *admissible*, meaning that for every v, $h(v) \leq \delta(v, t)$. The opposite is not always true.

Task 10.5. *Give an example of a consistent heuristic which causes A* to perform identically to Dijkstra's algorithm up until the point it visits the target.*

Task 10.6. *Suppose we use $h(v) = \delta(v, t)$. Argue that A* visits exactly the vertices on the shortest path between s and t, and no others. (Assume a unique shortest path from s to t.)*

Remark 10.7. *In the worst case, A* has the same asymptotic behavior as Dijkstra's algorithm – a good example of a graph for which A* performs no better than Dijkstra is a simple chain graph. However, for many kinds of graphs, A* is significantly faster than Dijkstra due to the vastly reduced number of vertices visited.*

Remark 10.8. *A simple and effective heuristic used in many applications is Euclidean distance. You should convince yourself that this heuristic is consistent.*

10.4   Additional Exercises

Exercise 10.9. *Design a consistent heuristic for the example graph given at the beginning of this recitation and run A* with that heuristic. Take note of how many vertices are left unvisited.*

Exercise 10.10. *Prove that for any consistent heuristic h, A* returns the shortest path between the source and target.*

Exercise 10.11. *Give an example of a small graph along with an admissible heuristic which is inconsistent.*

Exercise 10.12. *Prove that any consistent heuristic is also admissible.*

Exercise 10.13. *A simple modification of A* is called weighted-A*. In this algorithm, we take an additional parameter $w \geq 1.0$ and order the priority queue by $d(v) + w \cdot h(v)$. This has the tendency to make A* more directed towards the goal, further reducing the number of vertices visited at the cost of producing a potentially non-optimal solution.*

Prove that weighted-A* with weight w finds a path to the destination of length within a factor of w of the optimal path.

Exercise 10.14. *Implement Bellman-Ford in SML.*