Recitation 10

Shortest Paths

10.1 Announcements

- *ShortLab* has been released, and is due **Friday afternoon**. It's worth 125 points.
- *SegmentLab* will be released on **Friday**.

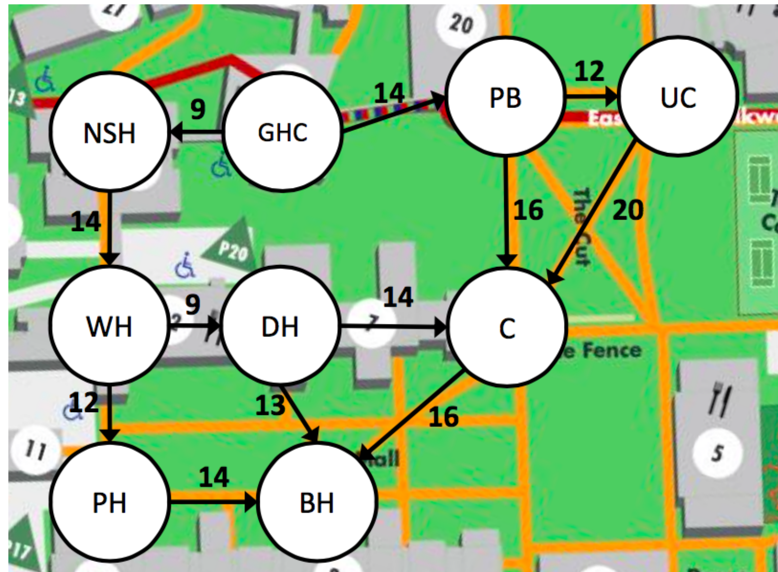
10.2 Dijkstra's Algorithm

For this section, we'll be using Dijkstra's algorithm as implemented in the textbook. It is also given here as a refresher.

Algorithm 10.1. *Dijkstra's Algorithm*

```
1  Dijkstra (G,s) =
2    let
3      dijkstra (X,Q) =
4        case PQ.deleteMin Q of
5          (NONE, _) ⇒ ⊥
6        | (SOME (d, v), Q') ⇒
7            if v ∈ X then dijkstra (X,Q') else
8            let
9              X' = X ∪ {v ↦ d}
10             relax (Q,(u,w)) = PQ.insert (Q,(d + w,u))
11             Q'' = iterate relax Q' (N+_G(v))
12           in
13             dijkstra (X',Q'')
14           end
15   in
16     dijkstra ({}, PQ.singleton (0,s))
17   end
```

Task 10.2. *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.*



Step	Recently Popped	Priority Queue
0	–	$\{(0, \texttt{GHC})\}$
1	GHC	$\{(9, \texttt{NSH}), (14, \texttt{PB})\}$
2	NSH	$\{(14, \texttt{PB}), (23, \texttt{WH})\}$
3	PB	$\{(23, \texttt{WH}), (26, \texttt{UC}), (30, \texttt{C})\}$
4	WH	$\{(26, \texttt{UC}), (30, \texttt{C}), (32, \texttt{DH}), (35, \texttt{PH})\}$
5	UC	$\{(30, \texttt{C}), (32, \texttt{DH}), (35, \texttt{PH}), (46, \texttt{C})\}$
6	C	$\{(32, \texttt{DH}), (35, \texttt{PH}), (46, \texttt{C}), (46, \texttt{BH})\}$
7	DH	$\{(35, \texttt{PH}), (45, \texttt{BH}), (46, \texttt{C}), (46, \texttt{BH}), (46, \texttt{C})\}$
8	PH	$\{(45, \texttt{BH}), (46, \texttt{C}), (46, \texttt{BH}), (46, \texttt{C}), (49, \texttt{BH})\}$
9	BH	$\{(46, \texttt{C}), (46, \texttt{BH}), (46, \texttt{C}), (49, \texttt{BH})\}$
10	C	$\{(46, \texttt{BH}), (46, \texttt{C}), (49, \texttt{BH})\}$
11	BH	$\{(46, \texttt{C}), (49, \texttt{BH})\}$
12	C	$\{(49, \texttt{BH})\}$
13	BH	$\{\}$

$$\{\texttt{GHC} \mapsto 0, \texttt{NSH} \mapsto 9, \texttt{PB} \mapsto 14, \texttt{WH} \mapsto 23, \texttt{UC} \mapsto 26, \texttt{C} \mapsto 30, \texttt{DH} \mapsto 32, \texttt{PH} \mapsto 35, \texttt{BH} \mapsto 45\}$$

Task 10.3. *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?

This modification has no effect on the amount of work performed by Dijkstra's algorithm in the worst case. For example, in a chain graph consisting of the directed edges $(i, i+1)$ for every $0 \leq i < n$, every time a vertex is inserted into the priority queue, it has not yet been visited.

Task 10.4. *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.

In the relax step, if a neighbor is already present in the priority queue, we perform a `decreaseKey` operation rather than an `insert`. We also only insert vertices which have not already been visited. In combination, these changes guarantee that every time we `deleteMin`, we get back a vertex which has not yet been visited. Notice, then, that

(a) for each vertex, we perform one `deleteMin` operation, and

(b) for each edge, we might either `insert` or `decreaseKey`.

Therefore the total running time is $O(m + n \log n)$, since `deleteMin` is logarithmic while `insert` and `decreaseKey` are constant-time.

Now, you might be concerned that it is expensive to keep track of what is and isn't in the priority queue. But, this is a non-issue: all we need is a set data structure with constant-time `insert`, `delete`, and `find` operations. For enumerated graphs, we can implement such a structure simply as an array of flags of length n (one flag for each vertex).

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 10.5. *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
```

To visualize the difference between using (target-specific) Dijkstra and A*, visit this link.

In order to ensure correctness, we need h to be *consistent*, meaning that for every two vertices u and v, $h(u) \leq \delta(u, v) + h(v)$ 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 10.6. *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.*

If $h(v) = 0$ for every v, then A*'s priority queue is ordered simply by $d(v)$, which is identical to Dijkstra's algorithm.

Task 10.7. *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.)*

Proof by induction. Let p be the sequence of vertices which forms the shortest path from s to t.

- At time 0, A* visits s, and obviously $p[0] = s$.
- At time $i > 0$, assume A* has visited exactly the vertices $p[0], \ldots, p[i-1]$. The vertex $p[i]$ must be in the priority queue with priority

$$d(p[i]) + \delta(p[i], t) = \delta(s, p[i-1]) + w(p[i-1], p[i]) + \delta(p[i], t) = \delta(s, t)$$

  and therefore $p[i]$ has the smallest priority in the priority queue (this should be straightforward to see via a proof by contradiction). A* will proceed by visiting $p[i]$.

Remark 10.8. *The runtime performance of A* depends heavily upon the quality of the heuristic. The previous two tasks highlight the extremes: the "worst" heuristic causes A* to do exactly the same thing as Dijkstra's, while the "best" causes it to visit only the nodes on the shortest path.*

Remark 10.9. *When searching a Euclidean space (where each vertex is a point in that space), a simple and effective heuristic is Euclidean distance. You should convince yourself that this heuristic is consistent.*

10.4 Additional Exercises

Exercise 10.10. *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 10.11. *Prove that for any consistent heuristic h, A* returns the shortest path between the source and target.*

Exercise 10.12. *Give an example of a small graph along with an admissible heuristic which is inconsistent.*

Exercise 10.13. *Prove that any consistent heuristic is also admissible.*

Exercise 10.14. *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.*

.