Lecture 14: Dijkstra's Algorithm

Today: SSSP without negative weights

- If all weights are ≥ 0, can do better than Bellman Ford's $O(|V| \cdot |E|)$
- Dijkstra's Algorithm takes $O(|E| + |V| \log |V|)$ time

Idea of Dijkstra's Algorithm

- Intuition: expanding frontier
  - At time x, identify all nodes with $\delta \leq x$
  - Frontier takes $w(u, v)$ time to traverse edge (u, v)
  - Grow frontier until all nodes are reached
- Familiar case:
  - If all weights are 1, this is BFS
  - For small integer weights, can subdivide edges and then use BFS
    * like Wleft Brothers in pset problem 6-4(a)
    * Positive integer weights $\leq k$ leads to $O(|V| + k \cdot |E|)$ runtime.
    * Bad if weights are big; doesn't work if weights are fractional
- Goal: Simulate continuous expansion process with discrete steps
  - Fast-forward to the next **event**: next time the frontier reaches a new vertex
  - Priority queue!
- Example (see slides)

Dijkstra's Algorithm

- Fits in relaxation framework
- Uses a Priority Queue to decide relaxation order
- Need a changeable Priority Queue! How?
  - Pset problem 5-2 showed how to cross-link a binary heap with a DAA (or Hash Table) to support `change_key` in $O(\log n)$ time (expected time if using a Hash Table)

    * Recall: `Q.change_key(id, new_key)` finds the item with identifier `id`, changes its key to `new_key`, and readjusts the queue as necessary
    * `decrease_key` is the special case where the new key must be smaller than the old key
  - Easier DAA-based priority queue can easily be modified in $O(1)$ time

```
def dijkstra_sssp(Adj, w, s):
    parent = [None] * len(Adj)   # Same
    parent[s] = s                # init
    d = [math.inf] * len(Adj)    # as
    d[s] = 0                     # before.

    Q = PriorityQueue.build(Item(id=u, key=d[u]) for u in Adj)

    while len(Q) > 0:
        u = Q.delete_min().id  # Delete and process u
        for v in Adj[u]:               # Same
            if d[v] > d[u] + w(u,v):   # relax
                d[v] = d[u] + w(u,v)   # as
                parent[v] = u          # before.
                Q.decrease_key(id=v, new_key=d[v]) # NEW!

    return d, parent
```

- Runtime is dominated by Priority Queue operations: `build` once, `delete_min` $|V|$ times, and `decrease_key` $|E|$ times.
- Which Priority Queue to use?! Four standard contenders (all runtimes are $O(\cdots)$):

Priority Queue	`build`	`del_min`	`dec_key`	Total
DAA Heap	V	V	1	V^2
Hash Table Heap	V ex.	V ex.	1 ex.	V^2 ex.
Binary Heap	V	$\log V$	$\log V$	$(V+E)\log V$
Fibonacci Heap	V	$\log V$ am.	1 am.	$E + V\log V$

- This class **does not cover** how Fibonacci Heaps work; you only need to know these runtimes
- When to use each?
  - Fibonacci Heap: Asymptotically best of the three options, but higher constant factors
  - Hash Table / Direct Access Array: Tied with Fibonacci Heaps for dense graphs with $E = \Omega(V^2)$.
  - Binary Heap: Tied with Fibonacci Heaps for sparse graphs with $E = O(V)$, e.g., planar graphs.
  - In the real world, Binary Heaps are often faster[citation needed], even if not asymptotically

Correctness of Dijkstra

- Intuition for the proof: the frontier includes nodes with "small" δ and excludes nodes with "large" δ. How to formalize this?

- Let B be the set of vertices that have been deleted from the Queue. Every loop adds one node to B.

- Loop Invariant:

  - For each node $u \in B$, $d[u] = \delta(s, u)$.
  - For each node $v \notin B$, its key $d[v]$ in Q equals the minimum weight of an s-v path that **only visits nodes in $\boldsymbol{B}$** except for v

- Base Case: True at beginning when $B = \emptyset$, since there's only one (zero-edge) path that stays in B until the last vertex, and that path has weight 0.

- Inductive step: assume B is not empty and satisfies our loop invariant. Let $M = \min\{d[v] \mid v \notin B\}$ be the minimum key in Q. (Note: Since keys are deleted in increasing order, we have $\delta(s, u) \leq M$ for every $u \in B$.)

- Claim: All paths from s that leave B have weight $\geq M$. Proof: Let x be the first node the path visits outside of B. Then the path has weight $\geq d[x] \geq M$.

- All nodes $v \notin B$ have $\delta(s, v) \geq M$. Proof: any s-v path leaves B.

- The next vertex u_{next} that is about to get deleted from Q has $d[u_{\text{next}}] = M$, since that is `delete_min`'s job.

- We just argued that $\delta(s, u_{\text{next}}) \geq M$, but by safety $\delta(s, u_{\text{next}}) \leq d[u_{\text{next}}] = M$, so indeed $d[u_{\text{next}}] = \delta(s, u_{\text{next}}) = M$.

- What about the rest of the nodes $v \notin B$ and $v \neq u_{\text{next}}$? What can the shortest s-v path that doesn't leave $B \cup \{u_{\text{next}}\}$ look like?

  - Case 1: the penultimate vertex is u_{next}. The best weight of such a path is $\delta(s, u_{\text{next}}) + w(u_{\text{next}}, v)$.
  - Case 2: the penultimate vertex is some $u \neq u_{\text{next}}$. Then replace the path to u with a shortest path that stays entirely in B, so this new path to v doesn't visit u_{next} at all. The smallest weight of such a path is (the old value of) $d[v]$.

- Since edge (u_{next}, v) gets relaxed when u_{next} is processed, the loop invariant is restored.

Single Pair Search: Compute a single $\delta(s, t)$

- What if we just need $\delta(s, t)$, don't care about the rest of the graph
- Idea: Search from both directions at once! When frontiers overlap, we've found the shortest path.
- Hopefully we've avoided looking at much of the graph.
- In more detail:
  - Run two Dijkstra's: one from s, and one backwards from t (with all edges reversed).
  - Two separate Priority Queues Q_s and Q_t, two parent pointers $\text{parent}_s[v]$ and $\text{parent}_t[v]$, and two distance measures $d_s[v]$ and $d_t[v]$
  - Interleave steps from the two Dijkstra runs, in any order
  - Stop after some vertex has been deleted and processed in both runs
  - **Warning**: This first vertex to be processed twice **might not** be on the shortest path!
  - Read through all processed vertices and find the vertex x that minimizes $d_s[x] + d_t[x]$ Use parent pointers to reconstruct and return the s-t path through x.
- Runtime: No guaranteed asymptotic gains, but often great in practice

SSSP Algorithm	Lecture	Setting	Running Time
BFS	10	$w(e) = 1$ for all $e \in E$	$O(V + E)$
DAG shortest paths	12	G acyclic	$O(V + E)$
Dijkstra	14	$w(e) \geq 0$ for all $e \in E$	$O(V \log V + E)$
Bellman-Ford	13	(general)	$O(VE)$

```python
def dijkstra_sssp(Adj, w, s):
    parent = [None] * len(Adj)  # Same
    parent[s] = s               # init
    d = [math.inf] * len(Adj)   # as
    d[s] = 0                    # before.

    Q = PriorityQueue.build(Item(id=u, key=d[u]) for u in Adj)

    while len(Q) > 0:
        u = Q.delete_min().id  # Delete and process u
        for v in Adj[u]:                # Same
            if d[v] > d[u] + w(u,v):    # relax
                d[v] = d[u] + w(u,v)    # as
                parent[v] = u           # before.
                Q.decrease_key(id=v, new_key=d[v]) # NEW!

    return d, parent
```

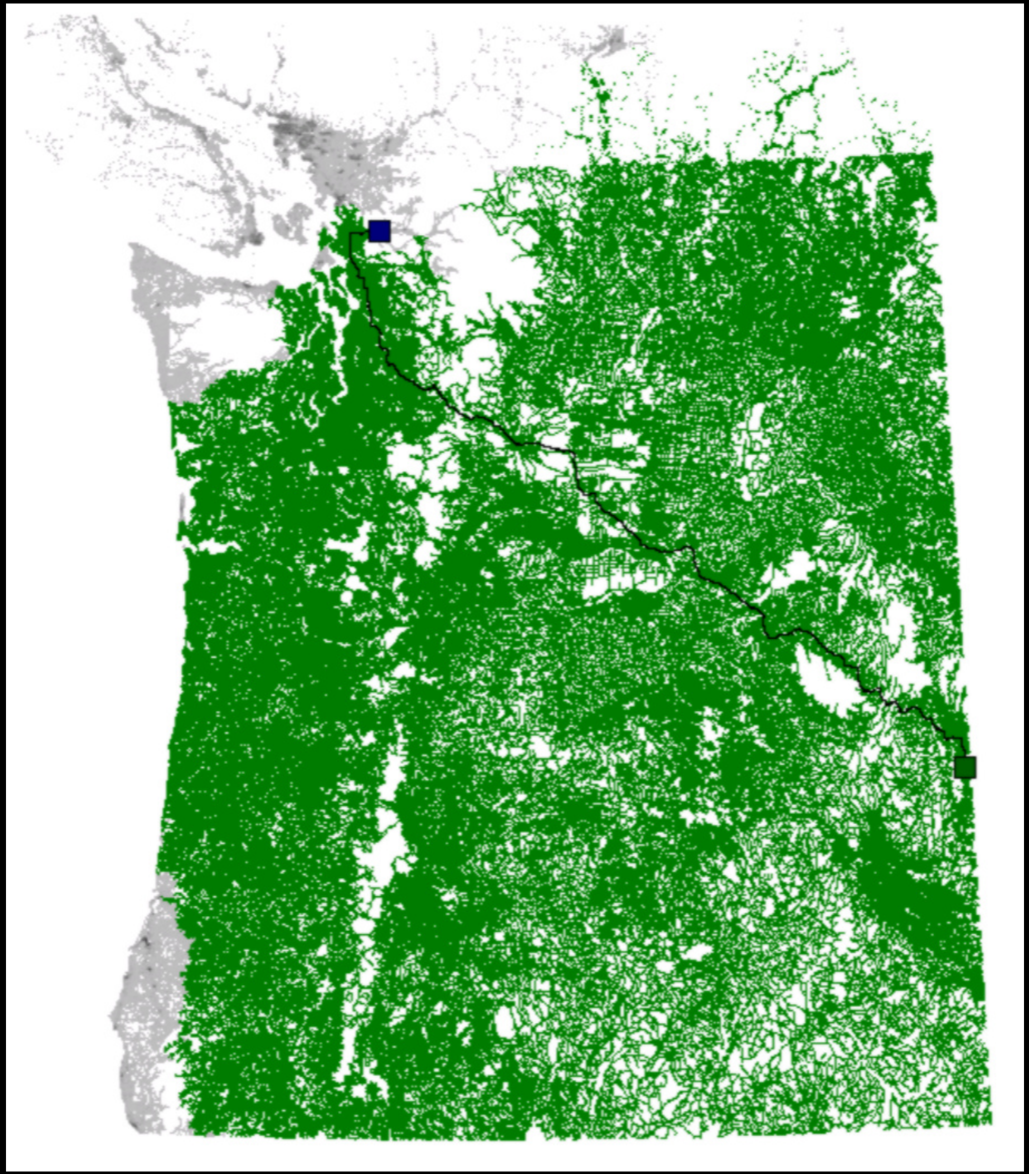
Dijkstra Runtime

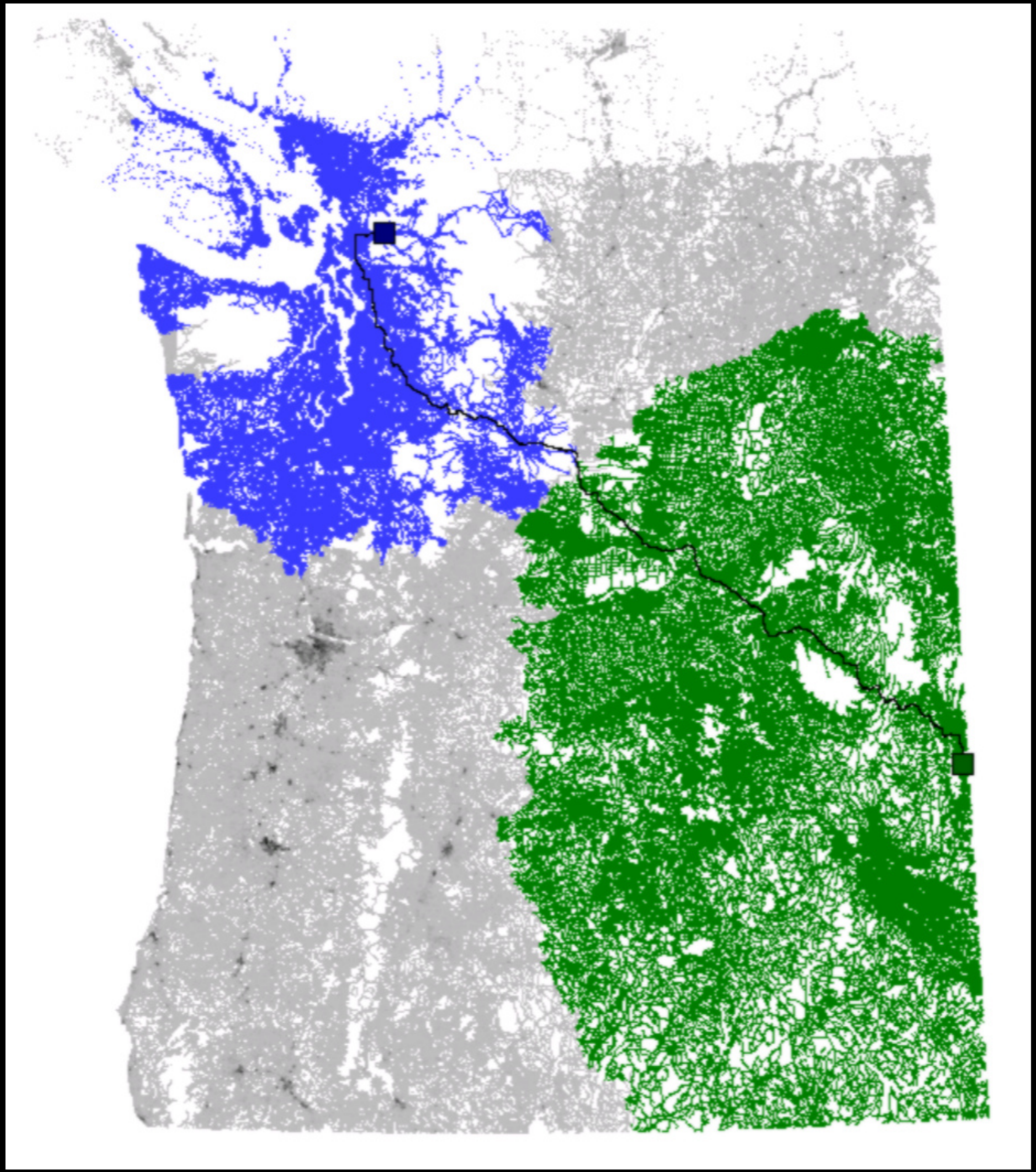
$$\text{Build} + V \cdot \text{DeleteMin} + E \cdot \text{DecreaseKey}$$

Queue	Build	Delete Min	Decrease Key	Total
DAA Heap	V	V	1	$V^2 + E = O(V^2)$
Hash Table Heap	V ex.	V ex.	1 ex.	V^2ex.
Bin Heap	V	$\log V$	$\log V$	$(V + E) \log V$
Fibonacci Heap	V	$\log V$ am.	**1 am.**	$V \log V + E$

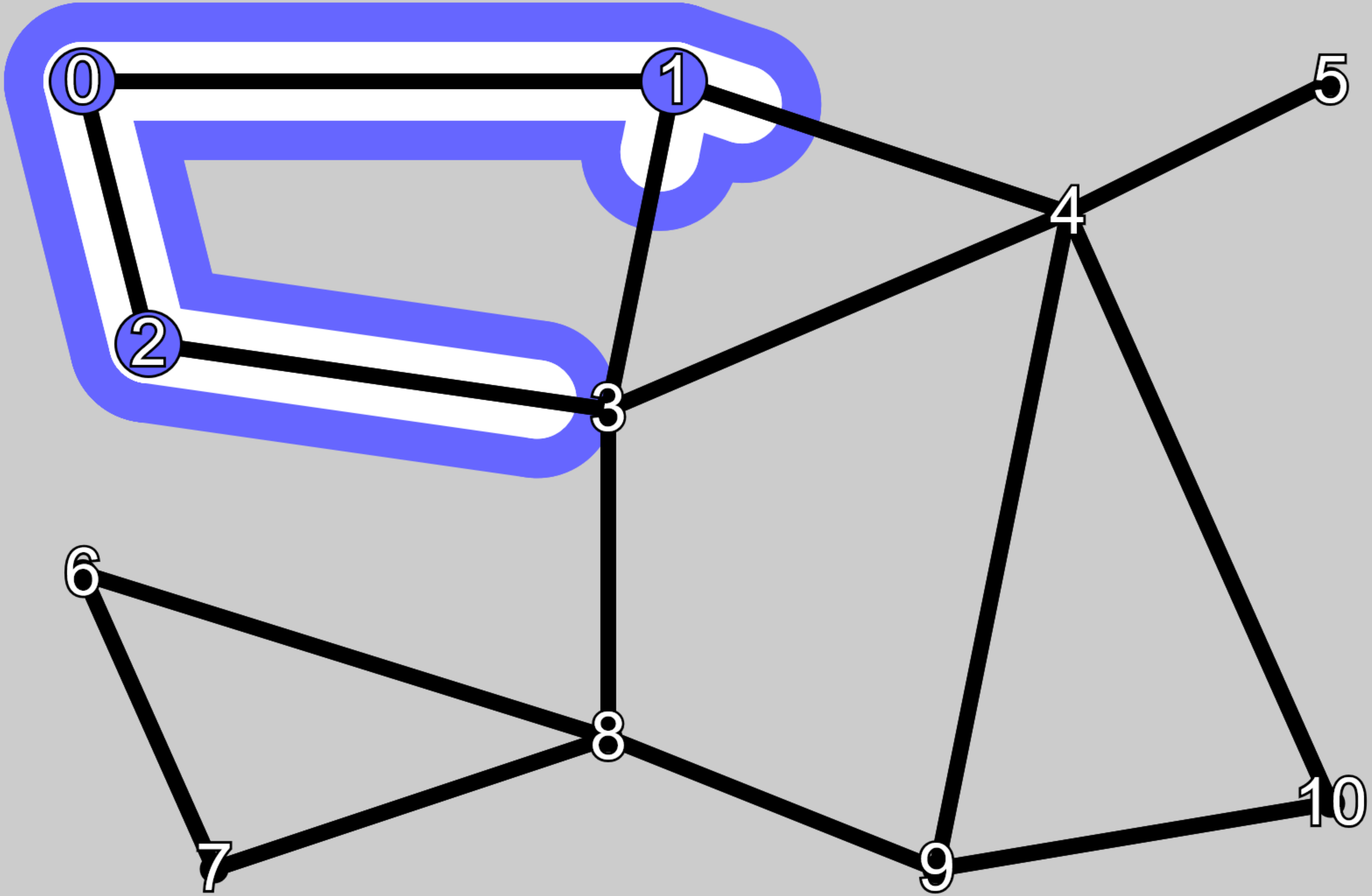
(All runtimes are big-O)

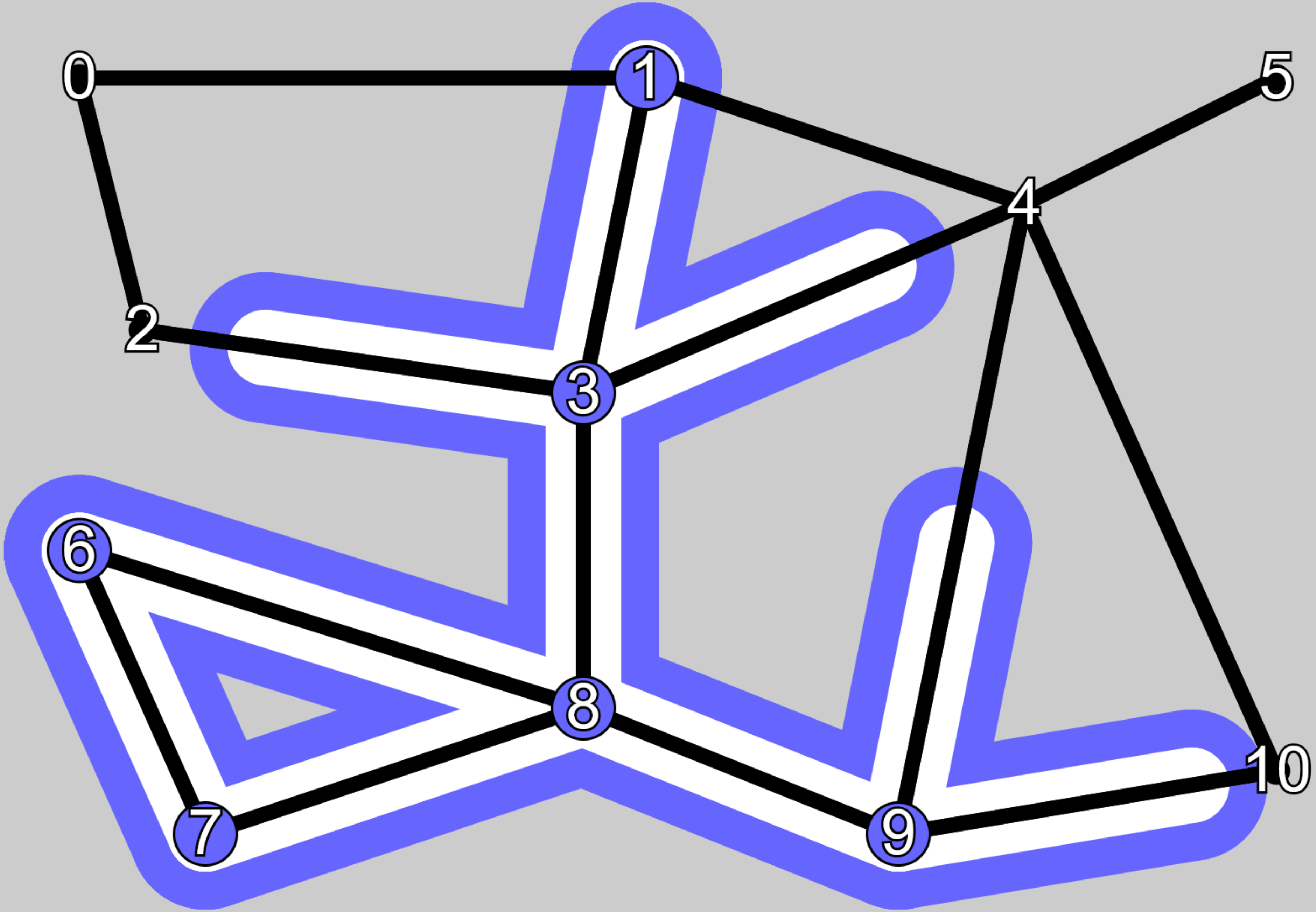
Dijkstra vs Bidirectional Dijkstra





http://www.columbia.edu/~cs2035/courses/ieor6614.S11/goldberg.pdf


0
1
2
3
4
5
6
7
8
9
10


0
1
2
3
4
5
6
7
8
9
10