Admin

Assignment 8 (DP coding). How did it go?

Assignment 9, graph algorithms: use/modify existing algorithms

Connectedness

Given an undirected graph, for every node $u \in V$, can we reach all other nodes in the graph?

Algorithm + running time

- Run BFS or DFS-Visit (one pass) and mark nodes as we visit them. If we visit all nodes, return true, otherwise false.

Running time: $O(|V| + |E|)$

Strongly connected

Given a directed graph, can we reach any node $v$ from any other node $u$?

Can we do the same thing?
Given a graph $G$, we can calculate the transpose of a graph $G^R$ by reversing the direction of all the edges.

Running time to calculate $G^R$? $\theta(|V| + |E|)$

**Strongly connected**

Strongly-Connected($G$)
- Run DFS/Visit or BFS from some node $u$
- If not all nodes are visited: return false
- Create graph $G^t$
- Run DFS/Visit or BFS on $G^t$ from node $u$
- If not all nodes are visited: return false
- return true

Is it correct?

What do we know after the first pass?
- Starting at $u$, we can reach every node

What do we know after the second pass?
- All nodes can reach $u$. Why?
- We can get from $u$ to every node in $G^t$, therefore, if we reverse the edges (i.e. $G$), then we have a path from every node to $u$

Which means that any node can reach any other node. Given any two nodes $s$ and $t$ we can create a path through $u$

$$s \rightarrow \ldots \rightarrow u \rightarrow \ldots \rightarrow t$$

Runtime?

Strongly-Connected($G$)
- Run DFS/Visit or BFS from some node $u$ $O(|V| + |E|)$
- If not all nodes are visited: return false $O(|V|)$
- Create graph $G^t$ $\theta(|V| + |E|)$
- Run DFS/Visit or BFS on $G^t$ from node $u$ $O(|V| + |E|)$
- If not all nodes are visited: return false $O(|V|)$
- return true

$O(|V| + |E|)$
Minimum spanning trees

What are they?
What do you remember about them?
What algorithms do you remember?

Input: An undirected, positive weight graph, $G=(V,E)$
Output: A tree $T=(V,E')$ where $E' \subseteq E$ that minimizes
$\text{weight}(T) = \sum_{e \in E'} w_e$

MST example

A
B
C
D
E
F

MSTs

Can an MST have a cycle?
MSTs

Can an MST have a cycle?

A MST is a minimum spanning tree, which is a tree that connects all vertices with the minimum total edge weight. An MST cannot have cycles.

Applications?

1. Connectivity
   - Networks (e.g. communications)
   - Circuit design/wiring
2. Hub/spoke models (e.g. flights, transportation)
3. Traveling salesman problem?

Algorithm ideas?

Cuts

A cut is a partitioning of the vertices into two sets $S$ and $V-S$.

An edge "crosses" the cut if it connects a vertex $u \in V$ and $v \in V-S$.
Minimum cut property
Given a partition $S$, let edge $e$ be the minimum cost edge that crosses the partition. Every minimum spanning tree contains edge $e$.

Prove this!

Consider an MST with edge $e'$ that is not the minimum edge.

Using $e$ instead of $e'$, still connects the graph, but produces a tree with smaller weights.

Minimum cut property
If the minimum cost edge that crosses the partition is not unique, then some minimum spanning tree contains edge $e$. 

Kruskal’s algorithm

Given a partition S, let edge \( e \) be the minimum cost edge that crosses the partition. Every minimum spanning tree contains edge \( e \).

\[
\text{Kruskal}(G) \\
1 \quad \text{for all } v \in V \\
2 \quad \text{MAKESET}(v) \\
3 \quad T \leftarrow \emptyset \\
4 \quad \text{sort the edges of } E \text{ by weight} \\
5 \quad \text{for all edges } (u, v) \in E \text{ in increasing order of weight} \\
6 \quad \text{if } \text{FIND-SET}(u) \neq \text{FIND-SET}(v) \\
7 \quad \text{add edge to } T \\
8 \quad \text{UNION}(	ext{FIND-SET}(u), \text{FIND-SET}(v))
\]
Kruskal’s algorithm

Add smallest edge that connects two sets not already connected

A
B
C
E
F
D

G

MST

B
D
F

A
C

1

2

3

4

5

6

25

A
B
C
E
F
D

G

MST

B
D
F

A
C

1

2

3

4

5

6

26

A
B
C
E
F
D

G

MST

B
D
F

A
C

1

2

3

4

5

6

27

A
B
C
E
F
D

G

MST

B
D
F

A
C

1

2

3

4

5

6

28
Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

G

MST

A
B
C
D
E
F

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

G

MST

A
B
C
D
E
F

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

G

MST

A
B
C
D
E
F

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

G

MST

A
B
C
D
E
F
Kruskal's algorithm

Add smallest edge that connects two sets not already connected

Kruskal's algorithm

Add smallest edge that connects two sets not already connected

G

MST

A
B
D
C
F

E

1
4
3
4
3
5
4
2
6

Kruskal's algorithm

Add smallest edge that connects two sets not already connected

A
B
D
C
F

E

1
4
3
5
4
2
6

Done!

Correctness of Kruskal's

Never adds an edge that connects already connected vertices

Always adds lowest cost edge to connect two sets. By min cut property, that edge must be part of the MST

Running time of Kruskal’s

Kruskal(G)
1 for all v \in V
2 \text{MAKESET}(v)
3 T = \{\}
4 sort the edges of E by weight
5 for all edges (u,v) \in E in increasing order of weight
6 if \text{FIND-SET}(u) \neq \text{FIND-SET}(v)
7 add edge to T
8 \text{UNION}(\text{FIND-SET}(u),\text{FIND-SET}(v))

33

34

35

36
Running time of Kruskal’s

\[
\begin{align*}
\text{Kruskal}(G) & \\
1 & \text{for all } v \in V \\
2 & \text{MakeSet}(v) \\
3 & \mathcal{P} = \emptyset \\
4 & \text{sort the edges of } E \text{ by weight} \\
5 & \text{for all edges } (x, y) \in E \text{ in increasing order of weight} \\
6 & \text{if } \text{FindSet}(x) \neq \text{FindSet}(y) \\
7 & \text{add edge } (x, y) \\
8 & \text{Union}(\text{FindSet}(x), \text{FindSet}(y)) \\
\end{align*}
\]

\[O(|E| \log |E|)\]

2 \(|E|\) calls to FindSet

|V| calls to Union

Disjoint set data structures

Represents a collection of one or more sets

Operations:
- **MakeSet**: Add a new value to the collections and make the value its own set
- **FindSet**: Given a value, return the set the value is in
- **Union**: Merge two sets into a single set

Disjoint set data structure

MakeSet(A), MakeSet(B), MakeSet(C), MakeSet(D), MakeSet(E)

Disjoint set data structure

FindSet(A)?

Disjoint Set

A  B  C  D  E

Disjoint Set

A  B  C  D  E
41. **Disjoint set data structure**

   - FindSet(A)?
   - Disjoint Set:
     - A
     - B
     - C
     - D
     - E

42. **Disjoint set data structure**

   - Union(FindSet(A), FindSet(E))
   - Disjoint Set:
     - A
     - B
     - C
     - D
     - E

43. **Disjoint set data structure**

   - Union(FindSet(A), FindSet(E))
   - Disjoint Set:
     - A
     - B
     - C
     - D
     - E

44. **Disjoint set data structure**

   - Union(FindSet(C), FindSet(D))
   - Disjoint Set:
     - A
     - B
     - C
     - D
     - E
Disjoint set data structure

Union(FindSet(C), FindSet(D))

Disjoint set data structure

FindSet(D)?

Disjoint set data structure

FindSet(D)?

Disjoint set data structure

Union(FindSet(D), FindSet(B))
Disjoint set data structure

Union(FindSet(D), FindSet(B))

How would we implement it with a list of linked lists?

MakeSet?

FindSet?

Union?
Disjoint set: union

53

Disjoint set: union

54

Disjoint set: union

Running time?

55

Disjoint set: union

O(1)

56
Disjoint set: find-set

- Search each linked list

Running time?

Disjoint set data structure

$$O(n) \quad n = \text{number of things in set}$$

Running time of Kruskal’s

| Disjoint set data structure | MakeSet (V calls) | FindSet (|E| calls) | Union (|V| calls) | Total |
|-----------------------------|-------------------|---------------------|------------------|-------|
| Linked lists                | $|V|$               | $O(|V| |E|)$         | $|V|$             | $O(|V||E| + |E| \log |E|)$ |
| Linked lists + heuristics   | $|V|$               | $O(|E| \log |V|)$   | $|V|$             | $O(|E| \log |V| + |E| \log |E|)$ |

O(|V| |E|)

O(|E| log |E|)
Prim's algorithm

Start at some root node and build out the MST by adding the lowest weighted edge at the frontier.

Prim(G, v)
1 for all v ∈ V
2 key[v] ← ∞
3 prev[v] ← null
4 key[v] ← 0
5 H ← MakeHeap(key)
6 while !Empty(H)
7 u ← Extract-Min(H)
8 visited[u] ← true
9 for each edge (u, v) ∈ E
10 if visited[v] and key[v] < key(u)
11 Decrease-Key(v, w(u, v))
12 prev[v] ← u
Prim's
6 while !Empty(H)
7 \textcolor{red}{u} \leftarrow \text{EXTRACT-MIN}(H)
8 \textcolor{red}{visited}[u] \leftarrow \text{true}
9 for each edge \langle u, v \rangle \in \text{E}
10 \textcolor{red}{\text{if } \text{heuristic}(u) \text{ and } w(u, v) < \text{key}(v)}
11 \textcolor{red}{\text{Decrease-Key}(v, w(u, v))}
12 \textcolor{red}{\text{prev}[v] \leftarrow u}

MST
6 while !Empty(H)
7 \textcolor{red}{u} \leftarrow \text{EXTRACT-MIN}(H)
8 \textcolor{red}{visited}[u] \leftarrow \text{true}
9 for each edge \langle u, v \rangle \in \text{E}
10 \textcolor{red}{\text{if } \text{heuristic}(u) \text{ and } w(u, v) < \text{key}(v)}
11 \textcolor{red}{\text{Decrease-Key}(v, w(u, v))}
12 \textcolor{red}{\text{prev}[v] \leftarrow u}
6 while not Empty(H):
7 u = extract-Min(H)
8 visited[u] = true
9 for each edge (u, v) ∈ E:
10 if visited[v] and w(u, v) < key[v]
11 decrease-key(v, w(u, v))
12 prev[v] ← u

6 while not Empty(H):
7 u = extract-Min(H)
8 visited[u] = true
9 for each edge (u, v) ∈ E:
10 if visited[v] and w(u, v) < key[v]
11 decrease-key(v, w(u, v))
12 prev[v] ← u

Nothing changes
**Correctness of Prim’s?**

Can we use the min-cut property?

- Given a partition $S$, let edge $e$ be the minimum cost edge that crosses the partition. Every minimum spanning tree contains edge $e$.

Let $S$ be the set of vertices visited so far.

The only time we add a new edge is if it’s the lowest weight edge from $S$ to $V \setminus S$.

---

**Running time of Prim’s**

```plaintext
Prim(G, r)
1 for all $v \in V$
2 $key[v] = \infty$
3 $key[r] = 0$
4 $H = \text{MakeHeap}(key)$
6 while $\text{Empty}(H)$
7 $u = \text{Extract-Min}(H)$
8 $\text{visited}[u] = \text{true}$
9 for each edge $(u, v) \in E$
10 if $\text{visited}[v]$ and $u(v, v) < key[v]$
11 $\text{Decrease-Key}(v, u(v, v))$
12 $\text{pre}[v] = u$
```
Running time of Prim’s

```
Prim(G, c)
1 for all v \in V
2 key[v] \leftarrow \infty
3 prev[v] \leftarrow null
4 h[0] = 0
5 H = Make-Heap(V, c)
6 while H is not empty
7 u \leftarrow \text{Extract-Min}(H)
8 for each edge (u, v) \in E
9 \quad \text{if } key[v] > key[u] + c(u, v)
10 \quad \text{key}[v] \leftarrow \text{key}[u] + c(u, v)
11 \quad prev[v] \leftarrow u
```

$\Theta(|V|)$

1 call to MakeHeap
$|V|$ calls to Extract-Min
$|E|$ calls to Decrease-Key

Array

1 MakeHeap $|V|$ ExtractMin $|V|$ DecreaseKey Total

Array

Bin heap

Fib heap

Kruskal’s:

Shortest paths

What is the shortest path from a to d?

BFS

Shortest paths
Shortest paths

What is the shortest path from a to d?

A

B

C

D

E

Shortest path algorithms?

Dijkstra’s algorithm

What is dist?

What is prev?

Dijkstra(G,a)
1 for all v in V
2 dist[v] ← ∞
3 prev[v] ← NULL
4 dist[a] ← 0
5 Q ← MstH(v)
6 while not.isEmpty(Q)
7 u ← ExtractMin(Q)
8 for all edges (u,v) in E
9 if dist[u] + w(u,v) > dist[v]
10 dist[v] ← dist[u] + w(u,v)
11 DecreaseKey(Q,v,dist[v])
12 prev[v] ← u

Shortest paths

What is the shortest path from a to d?

A

B

C

D

E

84

85

86

87
Dijkstra's algorithm

prev keeps track of the shortest path
**Dijkstra's algorithm**

**Dijkstra(G, s)**
1. for all \( v \in V \)
2. \( d[v] \leftarrow \infty \)
3. \( prev[v] \leftarrow null \)
4. \( d[s] = 0 \)
5. \( Q \leftarrow \text{MaxHeap}(V) \)
6. while \( \text{Empty}(Q) \)
7. \( u \leftarrow \text{ExtractMin}(Q) \)
8. for all \((u, v) \in E\)
   - \( d[u] \leftarrow \text{Dijkstra(G, u)} \)
9. \( d[s] = 0 \)
10. \( \text{DecreaseKey}(Q, u, d[u]) \)
11. \( prev[u] = s \)
12. \( d[u] = d[u] \)

**Heap**

- A 0
- B \( \infty \)
- C \( \infty \)
- D \( \infty \)
- E \( \infty \)
Dijkstra(G, e)
1 for all v ∈ V
2 dist[v] = ∞
3 prev[v] = null
4 dist[e] = 0
5 Q ← MaxPQ(V)
6 while ¬Empty(Q)
7 u ← ExtractMin(Q)
8 for all edges (u, v) ∈ E
9 if dist[u] + w(u, v) < dist[v]
10 dist[v] ← dist[u] + w(u, v)
11 DecreaseKey(Q, v, dist[v])
12 prev[v] ← u

Heap
A
B
C
C
D
D
E
E

Heap
B
C
D
D
E
E

Heap
C
1
B

Heap
C
1
B

Heap
C
1
B

Heap
C
1
B

Heap
C
1
B
**Dijkstra(G, s)**

1. for all v ∈ V
2. dist[v] ← ∞
3. prev[v] ← null
4. dist[s] ← 0
5. Q ← PriorityQueue
6. while Q is not empty
7. u ← ExtractMin(Q)
8. for all edges (u, v) ∈ E
9. if dist[v] > dist[u] + w(u, v)
10. dist[v] ← dist[u] + w(u, v)
11. DecreaseKey(Q, v, dist[v])
12. prev[v] ← u

**Heap**

```
E 3  D 5
```

---

**Dijkstra(G, s)**

1. for all v ∈ V
2. dist[v] ← ∞
3. prev[v] ← null
4. dist[s] ← 0
5. Q ← PriorityQueue
6. while Q is not empty
7. u ← ExtractMin(Q)
8. for all edges (u, v) ∈ E
9. if dist[v] > dist[u] + w(u, v)
10. dist[v] ← dist[u] + w(u, v)
11. DecreaseKey(Q, v, dist[v])
12. prev[v] ← u

**Heap**

```
D 5
```

---

**Dijkstra(G, s)**

1. for all v ∈ V
2. dist[v] ← ∞
3. prev[v] ← null
4. dist[s] ← 0
5. Q ← PriorityQueue
6. while Q is not empty
7. u ← ExtractMin(Q)
8. for all edges (u, v) ∈ E
9. if dist[v] > dist[u] + w(u, v)
10. dist[v] ← dist[u] + w(u, v)
11. DecreaseKey(Q, v, dist[v])
12. prev[v] ← u

**Heap**

```
B 1
```

---

**Dijkstra(G, s)**

1. for all v ∈ V
2. dist[v] ← ∞
3. prev[v] ← null
4. dist[s] ← 0
5. Q ← PriorityQueue
6. while Q is not empty
7. u ← ExtractMin(Q)
8. for all edges (u, v) ∈ E
9. if dist[v] > dist[u] + w(u, v)
10. dist[v] ← dist[u] + w(u, v)
11. DecreaseKey(Q, v, dist[v])
12. prev[v] ← u

**Heap**

```
D 5
```

---

**Dijkstra(G, s)**

1. for all v ∈ V
2. dist[v] ← ∞
3. prev[v] ← null
4. dist[s] ← 0
5. Q ← PriorityQueue
6. while Q is not empty
7. u ← ExtractMin(Q)
8. for all edges (u, v) ∈ E
9. if dist[v] > dist[u] + w(u, v)
10. dist[v] ← dist[u] + w(u, v)
11. DecreaseKey(Q, v, dist[v])
12. prev[v] ← u

**Heap**

```
B 1
```

---

**Dijkstra(G, s)**

1. for all v ∈ V
2. dist[v] ← ∞
3. prev[v] ← null
4. dist[s] ← 0
5. Q ← PriorityQueue
6. while Q is not empty
7. u ← ExtractMin(Q)
8. for all edges (u, v) ∈ E
9. if dist[v] > dist[u] + w(u, v)
10. dist[v] ← dist[u] + w(u, v)
11. DecreaseKey(Q, v, dist[v])
12. prev[v] ← u

**Heap**

```
B 1
```
Is Dijkstra’s algorithm correct?

Invariant: For every vertex removed from the heap, \( \text{dist}[v] \) is the actual shortest distance from \( s \) to \( v \)

1. The only time a vertex gets visited is when the distance from \( s \) to that vertex is smaller than the distance to any remaining vertex.
2. Therefore, there cannot be any other path that hasn’t been visited already that would result in a shorter path.

Running time?

\[
\text{Dijkstra}(G, s) \\
1 \text{ for all } v \in V \text{ do} \\
2 \quad \text{dist}(v) = \infty \\
3 \quad \text{prev}(v) = \text{null} \\
4 \quad \text{dist}(s) = 0 \\
5 \quad Q = \text{MaxHeap}(V) \\
6 \text{ while } \text{not Empty}(Q) \text{ do} \\
7 \quad u = \text{ExtractMax}(Q) \\
8 \quad \text{ for all edges } (u, v) \in E \text{ do} \\
9 \quad \quad \text{if dist}(v) > \text{dist}(u) + w(u, v) \text{ then} \\
10 \quad \quad \quad \text{dist}(v) = \text{dist}(u) + w(u, v) \\
11 \quad \quad \quad \text{prev}(v) = u \\
12 \text{ end while} \\
13 \text{return } \text{dist} \\
\]
Running time?

\texttt{Dijkstra}(G, s)
1. for all \( v \in V \)
   2. \( dists[v] \leftarrow \infty \)
   3. \( prev[v] \leftarrow \text{null} \)
4. \( dists[s] \leftarrow 0 \)
5. \( Q \leftarrow \text{MakeHeap}(V) \)

\textbf{1 call to MakeHeap}

\textbf{while} \( \text{not empty}(Q) \)
7. \( u \leftarrow \text{ExtractMin}(Q) \)
8. for all edges \((u, v) \in E\)
9. \( \text{if} \ ddist[u] > ddist[v] + w(u, v) \)
11. \( ddist[v] \leftarrow ddist[u] + w(u, v) \)
12. \( \text{DecrementKey}(Q, v, ddist[v]) \)
13. \( prev[v] \leftarrow u \)

\textbf{|V| calls}

Running time?

\texttt{Dijkstra}(G, s)
1. for all \( v \in V \)
   2. \( ddist[v] \leftarrow \infty \)
   3. \( prev[v] \leftarrow \text{null} \)
4. \( ddist[s] \leftarrow 0 \)
5. \( Q \leftarrow \text{MakeHeap}(V) \)

\textbf{[V] iterations}
6. \( u \leftarrow \text{ExtractMin}(Q) \)
7. for all edges \((u, v) \in E\)
8. \( \text{if} \ ddist[u] > ddist[v] + w(u, v) \)
10. \( ddist[v] \leftarrow ddist[u] + w(u, v) \)
11. \( \text{DecrementKey}(Q, v, ddist[v]) \)
12. \( prev[v] \leftarrow u \)

\textbf{|V| calls}

Running time?

\texttt{Dijkstra}(G, s)
1. for all \( v \in V \)
   2. \( ddist[v] \leftarrow \infty \)
   3. \( prev[v] \leftarrow \text{null} \)
4. \( ddist[s] \leftarrow 0 \)
5. \( Q \leftarrow \text{MakeHeap}(V) \)
6. \( u \leftarrow \text{ExtractMin}(Q) \)
7. for all edges \((u, v) \in E\)
8. \( \text{if} \ ddist[u] > ddist[v] + w(u, v) \)
10. \( ddist[v] \leftarrow ddist[u] + w(u, v) \)
11. \( \text{DecrementKey}(Q, v, ddist[v]) \)
12. \( prev[v] \leftarrow u \)

\textbf{|O(E)| calls}
### Running time?

<table>
<thead>
<tr>
<th>Depends on the heap implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MakeHeap</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Array</td>
</tr>
<tr>
<td>Bin heap</td>
</tr>
</tbody>
</table>

Is this an improvement? If \(|E| < |V|^2 / \log |V|\)

---

**What about Dijkstra's on...?**

```
\[ A \quad 10 \quad -10 \quad 5 \quad B \quad 1 \quad D \quad C \quad E \]
```
What about Dijkstra’s on...?

Dijkstra’s algorithm only works for positive edge weights

Is Dijkstra’s algorithm correct?

Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v

- The only time a vertex gets visited is when the distance from s to that vertex is smaller than the distance to any remaining vertex
- Therefore, there cannot be any other path that hasn’t been visited already that would result in a shorter path

We relied on having positive edge weights for correctness!

Dijkstra’s vs Prim’s

Dijkstra(G,s)
1 for all v ∈ V
2 dist[v] ← ∞
3 prev[v] ← null
4 Q ← MaxWeight(V)
5 while Q ≠ Ø
6 s ← ExtractMin(Q)
7 for all edges (u, v) ∈ E
8 if dist[u] + w(u, v) < dist[v]
9 dist[v] ← dist[u] + w(u, v)
10 prev[v] ← u

Prim(G,s)
1 for all v ∈ V
2 key[v] ← ∞
3 prev[v] ← null
4 Q ← MaxWeight(V)
5 while Q ≠ Ø
6 u ← ExtractMin(Q)
7 Q ← Q + Neighbors(u)
8 for each edge (u, v) ∈ E
9 if key[v] + w(u, v) < key[v]
10 key[v] ← key[v] + w(u, v)
11 prev[v] ← u