Weighted Graphs

Outline

• (single-source) shortest path
  – Dijkstra (Section 4.4)
  – Bellman-Ford (Section 4.6)

• (all-pairs) shortest path
  – Floyd-Warshall (Section 6.6)

• minimum spanning tree
  – Kruskal (Section 5.1.3)
  – Prim (Section 5.1.5)
Shortest Path

• Let $G$ be a weighted graph ($w(e)$ is the weight of the edge $e$)

• The length of a path $P$ is the sum of the weights of the edges of $P$

• If $P=e_0,e_1,...,e_{k-1}$ then the length of $P$ is $\sum w(e_i)$

Single-Source Shortest Path

• The distance from a vertex $u$ to vertex $v$, denoted by $d(u,v)$ is the length of a minimum length path (also called shortest-path) from $u$ to $v$, if such a path exists

• If the path does not exists, $d(u,v)=+\infty$

• Note that if there is a negative cycle, then the distance may not be defined
Optimal Substructure

- **Fact:** subpaths of shortest paths are shortest paths
- **Proof:** decompose a shortest path
  \[ p = <v_1, v_2, ..., v_k> \] into \( v_i \rightarrow v_i \rightarrow v_j \rightarrow v_k \). Then
  \[ w(p) = w(v_i, v_j) + w(v_j, v_k). \]
  If \( v_i \rightarrow v_j \) is not optimal, then we could make the path \( v_i \rightarrow v_k \) shorter, which contradicts the optimality of \( p \).

Shortest-Path Problems

- **Single-source (single-destination).** Find a shortest path from a given source (vertex \( s \)) to all the other vertices
  - positive weights → greedy algorithm
  - pos. & neg. weights → dynamic programming
- **All-pairs.** Find shortest-paths for every pair of vertices
  - pos. & neg. weights → dynamic programming
Dijkstra’s algorithm

• Dijkstra’s algorithm finds shortest paths from a start vertex \( s \) to all the other vertices

• It works on a simple graph with non-negative weights (i.e., it works only if \( w(e) \geq 0 \), for all edges \( e \))
Dijkstra’s algorithm

- The algorithm computes for each vertex $u$ the distance to $u$ from the start vertex $s$, that is, the weight of a shortest path between $s$ and $u$
- The algorithm keeps track of the set of vertices for which the distance has been computed, called the cloud $S$

Dijkstra’s algorithm

- Every vertex has a label associated with it
- For any vertex $u$, we can refer to its $d$ label as $d[u]$
- $d[u]$ stores an approximation of the distance between $s$ and $u$
- The algorithm will update a $d[u]$ value when it finds a shorter path from $s$ to $u$
Dijkstra’s algorithm

- When a vertex \( u \) is added to the cloud, its label \( d[u] \) is equal to the actual (final) distance between the starting vertex \( s \) and vertex \( u \).
- Initially, we set
  - \( d[s] = 0 \) ... the distance from \( s \) to itself is 0 ...
  - \( d[u] = \infty \) for \( u \neq s \) ... these will change ...

Edge relaxation

- For each vertex \( v \) in the graph, we maintain in \( d[v] \) the estimate of the shortest path from \( s \).
- Relaxing an edge \((u,v)\) means testing whether we can improve the shortest path to \( v \) found so far by going through \( u \).
Expanding the Cloud

- Repeat until all vertices have been put in the cloud
  - let $u$ be a vertex not in the cloud that has smallest $d[u]$
    (on the first iteration, the starting vertex will be chosen)
  - we add $u$ to the cloud $S$
  - we update $d[.]$ of the adjacent vertices of $u$ as follows
    \[(edge\ relaxation)\]
    \[\text{for each vertex } z \text{ adjacent to } u \text{ do}\]
    \[\text{if } z \text{ is not in the cloud } S \text{ then}\]
    \[\text{if } d[u] + \text{weight}(u,z) < d[z] \text{ then}\]
    \[d[z] \leftarrow d[u] + \text{weight}(u,z)\]

Example $s=\text{BWI}$
Example

Example
Example

Example
Dijkstra’s algorithm

**Algorithm** ShortestPath(G, v):

*Input:* A simple undirected weighted graph $G$ with nonnegative edge weights, and a distinguished vertex $v$ of $G$

*Output:* A label $D[u]$, for each vertex $u$ of $G$, such that $D[u]$ is the distance from $v$ to $u$ in $G$

Initialize $D[v] ← 0$ and $D[u] ← +\infty$ for each vertex $u \neq v$.

Let a priority queue $Q$ contain all the vertices of $G$ using the $D$ labels as keys.

**while** $Q$ is not empty **do**

  1. *{pull a new vertex $u$ into the cloud}*
  2. $u ← Q$.removeMin()
  3. **for** each vertex $z$ adjacent to $u$ such that $z$ is in $Q$ **do**

      *{perform the relaxation procedure on edge $(u,z)$}*

      **if** $D[u] + w((u,z)) < D[z]$ **then**

      $D[z] ← D[u] + w((u,z))$

      Change to $D[z]$ the key of vertex $z$ in $Q$.

  **end for**

**end while**

**return** the label $D[u]$ of each vertex $u$

---

**Time complexity**

- Use a *heap-based priority queue* $Q$ to store the vertices not in the cloud, where $d[u]$ is the key of a vertex $u$ in $Q$
- Insert all vertices in $Q$, takes $O(n \log n)$
- Each iteration of the while, we spend $O(\log n)$ time to remove vertex $u$ from $Q$ and $O(deg(u) \log n)$ to perform the relaxation step
- Overall, $O(n \log n + \sum_v(deg(v) \log n))$ which is $O((n+m) \log n)$ [using binary heaps]
Greedy choice

• **Theorem:** In Dijkstra’s algorithm, whenever a vertex \( u \) is pulled into \( S \), the label \( d[u] \) is equal to \( d(s,u) \) (the length of a shortest path from \( s \) to \( u \) ), and the equality is maintained thereafter.

• Proof: (by contradiction) omitted

Negative weights

• Dijkstra fails on graphs with negative edges.

• **Example:** Bringing \( z \) into \( C \) and performing edge relaxation invalidates the previously computed shortest path distance (124) to \( x \).
Bellman-Ford’s algorithm

- Dijkstra’s algorithm does not work when the weighted graph contains negative edges – we cannot be greedy anymore on the assumption that the lengths of paths will not decrease in the future
- Bellman-Ford’s algorithm detects negative cycles (returns false) or returns the shortest path-tree
Bellman-Ford’s algorithm

- Use $d[/]$ labels (like in Dijkstra’s and Prim’s)
- Initialize $d[s]=0$, $d[/]=\infty$ otherwise
- Perform $|V|-1$ rounds
- In each round, we attempt an edge relation for all the edges in the graph (arbitrary order)
- An extra round of edge relaxation can tell the presence of a negative cycle

Algorithm Bellman-Ford($G(V,E), s$)

```python
for each $u$ in $V$
    $d[u] \leftarrow \infty$
    $d[s] \leftarrow 0$

for $i \leftarrow 1$ to $|V|-1$ do
    for each $(u,v)$ in $E$ do
        if $d[v] > d[u] + w(u,v)$ then
            $d[v] \leftarrow d[u] + w(u,v)$

for each $(u,v)$ in $E$ do
    if $d[v] > d[u] + w(u,v)$ then
        return FALSE

return $d[\cdot], TRUE$
```
Iteration 0

Iteration 1
Iteration 2

Iteration 3
Observe that BF is essentially dynamic programming. Let \( d(i, j) = \) “cost of the shortest path from \( s \) to \( i \) that uses at most \( j \) edges/hops”

\[
d(i, j) = \begin{cases} 
0 & \text{if } i = s \text{ & } j = 0 \\
\infty & \text{if } i \neq s \text{ & } j = 0 \\
\min_{(k,i)} \{ d(k, j – 1) + w(k, i), d(i, j – 1) \} & \text{if } j > 0
\end{cases}
\]

Why \( O(nm) \)?
Bellman-Ford’s correctness

Theorem 7.4: If after performing the above computation there is an edge \((u,z)\) that can be relaxed (that is, \(D[u] + w((u,z)) < D[z]\)), then the graph \(G\) contains a negative-weight cycle. Otherwise, \(D[u] = d(v,u)\) for each vertex \(u\) in \(G\).

- Works for negative-weight edges
- Can detect the presence of negative-weight cycles
- Running time is \(O(nm)\)

Floyd-Warshall’s algorithm
All-pairs shortest path

- We want to compute the shortest path distance between every pair of vertices in a directed graph $G$ ($n$ vertices, $m$ edges)

- We want to know $D[i,j]$ for all $i,j$, where $D[i,j] =$ shortest distance from $v_i$ to $v_j$

All-pairs shortest path

- If $G$ has no negative-weight edges, we could use Dijkstra’s algorithm repeatedly from each vertex

- It would take $O(n (m+n) \log n)$ time, that is $O(n^2 \log n + nm \log n)$ time, which could be as large as $O(n^3 \log n)$
All-pairs shortest path

- If $G$ has negative-weight edges (but no negative-weight cycles) we could use Bellman-Ford’s algorithm repeatedly from each vertex
- Recall that Bellman-Ford’s algorithm runs in $O(nm)$
- It would take $O(n^2m)$ time, which could be as large $O(n^4)$ time

All-pairs shortest path

- We now see an algorithm to solve the all-pairs shortest path in $O(n^3)$ time
- The graph can contain negative-weight edges (but no negative-weight cycles)
All-pairs shortest path

- Let $G=(V,E)$ a weighted directed graph.
- Let $V=(v_1,v_2,...,v_n)$.
- Define cost function $D_{i,j}^k = \text{"the shortest distance from } v_i \text{ to } v_j \text{ using only vertices } \{v_1,v_2,...,v_k\}"$

A dynamic programming shortest-path

Initially we set

$$D_{i,j}^0 = \begin{cases} 0 & \text{if } i = j \\ w((v_i, v_j)) & \text{if } (v_i, v_j) \in E \\ +\infty & \text{otherwise} \end{cases}$$
A dynamic programming shortest-path

\[ w_{ij} \]

\[ v_i \rightarrow v_{ij} \rightarrow v_k \]

\[ \min\{v_{i \rightarrow v_{k, j}}, v_{i \rightarrow v_{k, i}}\} \]
A dynamic programming shortest-path

- The cost of going from $v_i$ to $v_j$ using vertices $1, ..., k$ is the shorter between
  - (do not to use $v_k$) The shortest path from $v_i$ to $v_j$ using vertices $1, ..., k-1$
  - (use $v_k$) The shortest path from $v_i$ to $v_k$ using $1, ..., k-1$ plus the cost of the shortest path from $v_k$ to $v_j$ using $1, ..., k-1$

Then

$$D_{i,j}^k = \min \{ D_{i,j}^{k-1}, D_{i,k}^{k-1} + D_{k,j}^{k-1} \}.$$
All-pairs shortest path

- Floyd-Warshall’s algorithm computes the shortest path distance between each pair of vertices of $G$ in $O(n^3)$ time

Minimum Spanning Tree
Minimum Spanning Tree

• Given a weighted undirected graph $G$, find a tree $T$ that spans all the vertices of $G$ and minimizes the sum of the weights on the edges, that is
  $$w(T) = \sum_{e \in T} w(e)$$

• We want a spanning tree of minimum cost

Example

\[ w(T) = 4 + 8 + 7 + 9 + 2 + 4 + 2 + 1 = 37 \]

Note that the MST is not necessarily unique

For example, add $(a, h)$, delete $(b, c)$
Growing a MST: Generic algorithm

• Grow MST one edge at a time
• Manage a set of edges $A$, maintaining the following invariant
  – prior to each iteration, $A$ is a subset of some MST
• At each iteration, we determine an edge $(u, v)$ that can be added to $A$ without violating this invariant
• If $A \cup \{(u, v)\}$ is also a subset of a MST, then $(u, v)$ is called a safe edge for $A$

Generic MST algorithm

\begin{verbatim}
GENERIC-MST(G, w)
1   A ← Ø
2   while A does not form a spanning tree
3       do find an edge $(u, v)$ that is safe for A
4           A ← A \cup \{(u, v)\}
5   return A
\end{verbatim}

• Loop in lines 2-4 is executed $|V| - 1$ times because any MST tree contains $|V| - 1$ edges
• The overall execution time depends on how to find a safe edge (step 3)
First Edge

• Which edge is clearly safe? Is the “shortest edge” safe?

Greedy Choice

• Definitions
  – Cut \((S, V-S)\): a partition of \(V\)
  – Crossing edge: one endpoint in \(S\) and the other in \(V-S\)
  – A cut respects a set of \(A\) of edges if no edges in \(A\) crosses the cut
  – A light edge crossing a partition if its weight is the minimum of any edge crossing the cut

• Theorem. Let \(A\) be a subset of \(E\) that is included in some MST of \(G=(V,E)\). Let \((S, V-S)\) be any cut of \(G\) that respects \(A\), and let \((u, v)\) be a light edge crossing \((S, V-S)\). Then, edge \((u, v)\) is safe for \(A\).
Proof of Greedy Choice Thm

- Let $T$ be a MST that includes $A$, and assume $T$ does not contain the light edge $(u, v)$. [If it does, we are done.]

- First, we construct another MST $T'$ that includes $A \cup \{(u, v)\}$
  Adding $(u, v)$ to $T$ induces a cycle
  - Let $(x, y)$ be the edge on the cycle crossing $(S, V-S)$, then $w(u, v) \leq w(x, y)$
  - $T' = T - (x, y) \cup (u, v)$
  - $T'$ is also a MST because it is a spanning tree of $G$ and $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$

- Second, we prove that $(u, v)$ is safe for $A$
  - Since $A \subseteq T$ and $(x, y) \notin A$ then $A \subseteq T'$. Therefore $A \cup \{(u, v)\} \subseteq T'$. Since $T'$ is a MST, $(u, v)$ is safe for $A$
Optimal substructure property

- Let $T$ be an MST of $G$. Let $(u,v)$ be an edge in $T$
- Removing $(u,v)$ partitions $T$ into two trees $T_1$ and $T_2$
- Let $(S, V-S)$ be a cut that respect $T$, let $E_1$ be the subset of edges incident to $S$, and $E_2$ be the subset of edges incident to $V-S$
- Claim: $T_1$ is an MST of $G_1 = (S, E_1)$, and $T_2$ is an MST of $G_2 = (V-S, E_2)$
  - Note that $w(T) = w(u,v) + w(T_1) + w(T_2)$
  - A “cheaper” tree than $T_1$ or $T_2$ cannot exists, otherwise $T$ would not be optimal

Generic MST algorithm

```
GENERIC-MST(G, w)
1    A ← ∅
2    while A does not form a spanning tree
3        do find an edge $(u, v)$ that is safe for $A$
4            A ← A ∪ {(u, v)}
5    return A
```
Kruskal’s algorithm

- Consider the edges one at a time, by increasing weight

- Accept an edge if it connects two different trees
Example

(a)

(b)

Example

(c)

(d)
Example

(e)

(f)

Example

(g)

(h)
Example

(i)

Example

(k)

(ii)

Example
Example

Kruskal’s algorithm

**Algorithm** Kruskal(G):

**Input:** A simple connected weighted graph G with \( n \) vertices and \( m \) edges

**Output:** A minimum spanning tree \( T \) for \( G \)

for each vertex \( v \) in \( G \) do

- Define an elementary cluster \( C(v) \) \( \leftarrow \{v\} \).
- Initialize a priority queue \( Q \) to contain all edges in \( G \), using the weights as keys.

\( T \leftarrow \emptyset \) \( \{T \) will ultimately contain the edges of the MST\}

while \( T \) has fewer than \( n - 1 \) edges do

- \((u,v) \leftarrow Q\text{.removeMin}()\)
- Let \( C(v) \) be the cluster containing \( v \), and let \( C(u) \) be the cluster containing \( u \).

if \( C(v) \neq C(u) \) then

- Add edge \((v,u)\) to \( T \).
- Merge \( C(v) \) and \( C(u) \) into one cluster, that is, union \( C(v) \) and \( C(u) \).

return tree \( T \)
Data Structure for Kruskal’s algorithm

- The data structure maintains a forest of trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with the following operations
  - \textit{find}(u): return the set storing \( u \)
  - \textit{union}(u, v): replace the sets storing \( u \) and \( v \) with their union

Data structure for sets

A={1,4,7}  B={2,3,6,9}  C={5,8,10,11,12}
Representation of a Partition

- Each set is stored in a sequence (list)
- Each element has a reference back to the set
  - operation $\text{find}(u)$ takes $O(1)$ time, and returns the set of which $u$ is a member
  - in operation $\text{union}(u,v)$, we move the elements of the smaller set to the sequence of the larger set and update their references
  - the time for operation $\text{union}(u,v)$ is $\min(n_u,n_v)$, where $n_u$ and $n_v$ are the sizes of the sets storing $u$ and $v$

Kruskal’s algorithm running time

- Whenever a vertex is added to a tree, the size of the tree containing the vertex at least double
- Each vertex is moved to a new tree at most $\log n$ times
- Total time merging trees is $O(n \log n)$
- Cost of creating the priority queue $O(m \log m)$ which is $O(m \log n)$
- Overall running time is $O((n+m) \log n)$
Prim’s algorithm

- The edges in the set $A$ always form a single tree
- The tree starts from an arbitrary vertex and grows until the tree spans all the vertices in $V$
- At each step, a light edge is added to the tree $A$ that connects $A$ to an isolated vertex of $G_A=(V, A)$
- “Greedy” because the tree is augmented at each step with an edge that contributes the minimum amount possible to the tree’s weight
Prim’s vs. Dijkstra’s

• Prim’s strategy similar to Dijkstra’s
• Grows the MST $T$ one edge at a time
• Cloud covering the portion of $T$ already computed
• Label $D[u]$ associated with each vertex $u$ outside the cloud (distance to the cloud)

Prim’s algorithm

• For any vertex $u$, $D[u]$ represents the weight of the current best edge for joining $u$ to the rest of the tree in the cloud (as opposed to the total sum of edge weights on a path from start vertex to $u$)
• Use a priority queue $Q$ whose keys are $D$ labels, and whose elements are vertex-edge pairs
Prim’s algorithm

- Any vertex $v$ can be the starting vertex
- We still initialize $D[v]=0$ and all the $D[u]$ values to $+\infty$
- We can reuse code from Dijkstra’s, just change a few things

Example
Example

\( (c) \)

\( (d) \)

Example

\( (e) \)

\( (f) \)
Example

Example

(i)

(j)
Pseudo Code

Algorithm PrimJarnik(G):

Input: A weighted connected graph G with n vertices and m edges
Output: A minimum spanning tree T for G

1. Pick any vertex v of G
2. D[v] ← 0
3. for each vertex u ≠ v do
   4. D[u] ← ∞
   5. Initialize T ← ∅.
   6. Initialize a priority queue Q with an item ((u, null), D[u]) for each vertex u, where (u, null) is the element and D[u] is the key.
   7. while Q is not empty do
      8. (u, e) ← Q.removeMin()
      9. Add vertex u and edge e to T.
      10. [perform the relaxation procedure on edge (u, z)]
          if w((u, z)) < D[z] then
             11. D[z] ← w((u, z))
             12. Change to (z, (u, z)) the element of vertex z in Q.
             13. Change to D[z] the key of vertex z in Q.

14. return the tree T

Time complexity

- Initializing the queue takes \(O(n \log n)\) [binary heap]
- Each iteration of the while, we spend \(O(\log n)\) time to remove vertex \(u\) from \(Q\) and \(O(\text{deg}(u) \log n)\) to perform the relaxation step
- Overall, \(O(n \log n + \sum\text{deg}(v) \log n)\) which is \(O((n+m) \log n)\) if using a binary heap
Summary

<table>
<thead>
<tr>
<th></th>
<th>Time complexity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>$O((n+m) \log n)$ using p.q.</td>
<td>Non-negative weights</td>
</tr>
<tr>
<td></td>
<td>$O(n^2+m)$ using array</td>
<td></td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>$O(mn)$</td>
<td>Negative weights ok</td>
</tr>
<tr>
<td>All-pairs</td>
<td>$O(n^2)$</td>
<td>Negative weights ok</td>
</tr>
<tr>
<td>Kruskal</td>
<td>$O((n+m) \log n)$ using p.q.</td>
<td></td>
</tr>
<tr>
<td>Prim-Jarnik</td>
<td>$O((n+m) \log n)$ using p.q.</td>
<td></td>
</tr>
</tbody>
</table>

Reading Assignment

- Dasgupta
  - single-source shortest path (4.4, 4.6 and 4.7)
  - all-pairs shortest path (6.6)
  - minimum spanning tree (5.1.3, 5.1.5)