A General View of Graph Search

Goals:
• Explore the graph systematically starting at $s$ to
• Find a vertex $t$ / Find a path from $s$ to $t$.
• Find the shortest path from $s$ to all vertices.
• …
A General View of Graph Search

In every step of the search we maintain
- The part of the graph already explored.
- The part of the graph not yet explored.
- A data structure (an agenda) of next edges (adjacent to the explored graph).

 Agenda: \((v2,v5), (v4,v5), (v4,v7)\)
The graph search algorithms discussed so far differ almost only in the type of agenda they use:

- **DFS**: uses a stack.
- **BFS**: uses a queue.
- **Dijkstra’s**: uses a priority queue.
- **Topological Sort**: BFS with constraint on items in the queue.

**Agenda:** \((v_2, v_5), (v_4, v_5), (v_4, v_7)\)
Correctness of Dijkstra’s Algorithm

• We want to show that Dijkstra’s algorithm really finds the minimum path costs (we don’t miss any shorter solutions by choosing the shortest edge greedily).

• Proof by induction on the set S of visited nodes.

• Base case:
  |S|=1. Trivial. Length shortest path is 0.
Correctness of Dijkstra’s Inductive Step

• Assume the algorithm produces the minimal path cost from s for the subset S, |S| = k.

• Dijkstra’s algorithm selects the next edge (u,v) leaving S.

• Assume there was a shorter path from s to v that does not contain (u,v).
  • Then that path must contain another edge (x,y) leaving S.
  • The cost of (x,y) is already higher than (u,v) because we didn’t choose it before (u,v)
  • Therefore (u,v) must be on the shortest path.
Designing a Home Network.

- BR 1
- living room
- kitchen
- dining room
- BR 2
- office
- BR 3
- Attic
- basement
- garage
Designing a Home Network.
Designing a Home Network.

Total cost: 62
Designing a Home Network.

Total cost: 44
Designing a Home Network.

Total cost: 32
Spanning Trees

• Given an undirected, connected graph $G=(V,E)$.

• A **spanning tree** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$
Spanning Trees

- Given an undirected, connected graph $G=(V,E)$.

- A **spanning tree** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$

$T$ is acyclic. There is a single path between any pair of vertices.
Spanning Trees

• Given an undirected, connected graph $G=(V,E)$.

• A **spanning tree** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$

$T$ is acyclic. There is a single path between any pair of vertices.
Spanning Trees

• Given an undirected, connected graph \( G=(V,E) \).

• A **spanning tree** is a tree that connects all vertices in the graph. \( T=(V, E_T \subseteq E) \)

Any node can be the root of the spanning tree.
Spanning Trees

• Given an undirected, connected graph $G=(V,E)$.

• A **spanning tree** is a tree that connects all vertices in the graph. $T=(V, E_T \subseteq E)$

Number of edges in a spanning tree: $|V|-1$
Spanning Trees, Applications

- Constructing a computer/power networks (connect all vertices with the smallest amount of wire).
- Clustering Data.
- Dependency Parsing of Natural Language (directed graphs. This is harder).
- Constructing mazes.
- ...
- Approximation algorithms for harder graph problems.
- ...
Minimum Spanning Trees

• Given a *weighted* undirected graph $G=(E,V)$.

• A *minimum spanning tree* is a spanning tree with the minimum sum of edge weights.
Minimum Spanning Trees

• Given a \textit{weighted} undirected graph $G=(E,V)$.

• A \textbf{minimum spanning tree} is a spanning tree with the minimum sum of edge weights.

Total cost = 16

(often there are multiple minimum spanning trees)
Prim’s Algorithm for finding MSTs

• Another greedy algorithm. A variant of Dijkstra’s algorithm.

• Cost annotations for each vertex v reflect the lowest weight of an edge connecting v to other vertices already visited.
  • That means there might be a lower-weight edge from another vertices that have not been seen yet.

• Keep vertices on a priority queue and always expand the vertex with the lowest cost annotation first.
Prim’s Algorithm

Use a Priority Queue $q$
- for all $v \in V$
  set $v.cost = \infty$, set $v.visited = \text{false}$
- Choose any vertex $s$.
  set $s.cost = 0$, $s.visited = \text{true}$;
- $q.insert(s)$
- While $q$ is not empty:
  - $(costu, u) <- q.deleteMin()$
  - if not $u.visited$:
    - $u.visited = \text{True}$
    - for each edge $(u,v)$:
      - if not $v.visited$:
        - if $(cost(u,v) < v.cost)$
          - $v.cost = cost(u,v)$
          - $v.parent = u$
          - $q.insert((v.cost,v))$
Prim’s Algorithm

Use a Priority Queue $q$
- for all $v \in V$
  - set $v.cost = \infty$, set $v.visited = false$
- Choose any vertex $s$
  - set $s.cost = 0$, $s.visited = true$;
- $q.insert(s)$

- While $q$ is not empty:
  - $(cost_u, u) <- q.deleteMin()$
  - if not $u.visited$:
    - $u.visited = True$
  - for each edge $(u,v)$:
    - if not $v.visited$:
      - if $(cost(u,v) < v.cost)$
        - $v.cost = cost(u,v)$
        - $v.parent = u$
        - $q.insert((v.cost,v))$
Prim’s Algorithm

Use a Priority Queue \( q \)
- for all \( v \in V \)
  - set \( v.cost = \infty \), set \( v.visited = \text{false} \)
- Choose any vertex \( s \).
  - set \( s.cost = 0 \), \( s.visited = \text{true} \);
- \( q.insert(s) \)
- While \( q \) is not empty:
  - \((\text{cost}_u, u) \leftarrow q.deleteMin()\)
  - if not \( u.visited \):
    - \( u.visited = \text{True} \)
    - for each edge \((u,v)\):
      - if not \( v.visited \):
        - if \((\text{cost}(u,v) < v.cost)\)
          - \( v.cost = \text{cost}(u,v) \)
          - \( v.parent = u \)
          - \( q.insert((v.cost,v)) \)
Prim’s Algorithm

Use a Priority Queue \( q \)
- for all \( v \in V \)
  
  set \( v.\text{cost} = \infty \), set \( v.\text{visited} = \text{false} \)
- Choose any vertex \( s \).
  
  set \( s.\text{cost} = 0 \), \( s.\text{visited} = \text{true} \);
- \( q.\text{insert}(s) \)

- While \( q \) is not empty:
  - \((\text{cost}_u, u) \leftarrow q.\text{deleteMin}()\)
  - if not \( u.\text{visited} \):
    - \( u.\text{visited} = \text{True} \)
    - for each edge \((u,v)\):
      - if not \( v.\text{visited} \):
        - if \((\text{cost}(u,v) < v.\text{cost})\)
          - \( v.\text{cost} = \text{cost}(u,v) \)
          - \( v.\text{parent} = u \)
          - \( q.\text{insert}((v.\text{cost}, v)) \)
Prims’s Algorithm

Use a Priority Queue $q$
- for all $v \in V$
  - set $v$.cost = $\infty$, set $v$.visited = false
- Choose any vertex $s$.
  - set $s$.cost = 0, $s$.visited = true;
- $q$.insert($s$)

- While $q$ is not empty:
  - $(\text{cost}_u, u) \leftarrow q$.deleteMin()
  - if not $u$.visited:
    - $u$.visited = True
    - for each edge $(u,v)$:
      - if not $v$.visited:
        - if ($\text{cost}(u,v) < v$.cost)
          - $v$.cost = $\text{cost}(u,v)$
          - $v$.parent = $u$
          - $q$.insert($((v$.cost,$v))$)
Use a Priority Queue $q$

- for all $v \in V$
  - set $v\.cost = \infty$, set $v\.visited = \text{false}$
- Choose any vertex $s$.
  - set $s\.cost = 0$, $s\.visited = \text{true}$;
- $q\.insert(s)$

- While $q$ is not empty:
  - $(\text{cost}u, u) \leftarrow q\.deleteMin()$
  - if not $u\.visited$:
    - $u\.visited = \text{True}$
    - for each edge $(u,v)$:
      - if not $v\.visited$:
        - if $(\text{cost}(u,v) < v\.cost)$
          - $v\.cost = \text{cost}(u,v)$
          - $v\.parent = u$
          - $q\.insert((v\.cost, v))$
Prim’s Algorithm

Use a Priority Queue $q$
- for all $v \in V$
  set $v.cost = \infty$, set $v.visited = \text{false}$
- Choose any vertex $s$.
  set $s.cost = 0$, $s.visited = \text{true}$;
- $q.insert(s)$
- While $q$ is not empty:
  - $(\text{cost}_u, u) \leftarrow q.deleteMin()$
  - if not $u.visited$:
    - $u.visited = \text{True}$
    - for each edge $(u,v)$:
      - if not $v.visited$:
        - if $(\text{cost}(u,v) < v.cost)$
          - $v.cost = \text{cost}(u,v)$
          - $v.parent = u$
          - $q.insert((v.cost,v))$
Prim’s Algorithm

Use a Priority Queue $q$

- for all $v \in V$
  - set $v.cost = \infty$, set $v.visited = false$
- Choose any vertex $s$.
  - set $s.cost = 0$, $s.visited = true$;
- $q.insert(s)$

- While $q$ is not empty:
  - $(cost_u, u) < q.deleteMin()$
  - if not $u.visited$:
    - $u.visited = True$
    - for each edge $(u,v)$:
      - if not $v.visited$:
        - if $(cost(u,v) < v.cost)$
          - $v.cost = cost(u,v)$
          - $v.parent = u$
          - $q.insert((v.cost,v))$
Prim’s Algorithm

Use a Priority Queue $q$
- for all $v \in V$
  set $v\.cost = \infty$, set $v\.visited = \text{false}$
- Choose any vertex $s$.
  set $s\.cost = 0$, $s\.visited = \text{true}$;
- $q\.insert(s)$

- While $q$ is not empty:
  - $(\text{cost}_u, u) \leftarrow q\.deleteMin()$
  - if not $u\.visited$:
    - $u\.visited = \text{True}$
    - for each edge $(u,v)$:
      - if not $v\.visited$:
        - if $(\text{cost}(u,v) < v\.cost)$
          - $v\.cost = \text{cost}(u,v)$
          - $v\.parent = u$
          - $q\.insert((v\.cost,v))$

Running time: Same as Dijkstra’s Algorithm $O(|E| \log |V|)$
Kruskal’s Algorithm for finding MSTs

• Kruskal’s algorithm maintains a “forest” of trees.
• Initially each vertex is its own tree.
• Sort edges by weight. Then attempt to add them one-by-one. Adding an edge merges two trees into a new tree.
• If an edge connects two nodes that are already in the same tree it would produce a cycle. Reject it.
Kruskal’s Algorithm

Sort edges (or keep them on a heap)

(v1,v2) 2
(v1,v3) 4
(v1,v4) 1
(v2,v4) 3
(v2,v5) 10
(v3,v4) 2
(v3,v6) 4
(v4,v5) 7
(v4,v6) 8
(v4,v7) 4
(v5,v7) 6
(v6,v7) 1
Kruskal’s Algorithm

(v1,v4) 1
(v6,v7) 1
(v1,v2) 2
(v3,v4) 2
(v2,v4) 3
(v1,v3) 4
(v3,v6) 4
(v4,v7) 4
(v5,v7) 6
(v4,v5) 7
(v4,v6) 8
(v2,v5) 10
Kruskal’s Algorithm

(v1, v4) 1
(v6, v7) 1
(v1, v2) 2
(v3, v4) 2
(v2, v4) 3
(v1, v3) 4
(v3, v6) 4
(v4, v7) 4
(v5, v7) 6
(v4, v5) 7
(v4, v6) 8
(v2, v5) 10
Kruskal’s Algorithm

(v1,v4) 1  OK
(v6,v7) 1  OK
(v1,v2) 2
(v3,v4) 2
(v2,v4) 3
(v1,v3) 4
(v3,v6) 4
(v4,v7) 4
(v5,v7) 6
(v4,v5) 7
(v4,v6) 8
(v2,v5) 10
Kruskal’s Algorithm

\[ (v_1,v_4) \quad 1 \quad \text{OK} \]
\[ (v_6,v_7) \quad 1 \quad \text{OK} \]
\[ (v_1,v_2) \quad 2 \quad \text{OK} \]
\[ (v_3,v_4) \quad 2 \]
\[ (v_2,v_4) \quad 3 \]
\[ (v_1,v_3) \quad 4 \]
\[ (v_3,v_6) \quad 4 \]
\[ (v_4,v_7) \quad 4 \]
\[ (v_5,v_7) \quad 6 \]
\[ (v_4,v_5) \quad 7 \]
\[ (v_4,v_6) \quad 8 \]
\[ (v_2,v_5) \quad 10 \]
Kruskal’s Algorithm

(v1,v4) 1  OK
(v6,v7) 1  OK
(v1,v2) 2  OK
(v3,v4) 2  OK
(v2,v4) 3
(v1,v3) 4
(v3,v6) 4
(v4,v7) 4
(v5,v7) 6
(v4,v5) 7
(v4,v6) 8
(v2,v5) 10
Kruskal’s Algorithm

(v1,v4) 1 OK
(v6,v7) 1 OK
(v1,v2) 2 OK
(v3,v4) 2 OK
(v2,v4) 3 reject
(v1,v3) 4
(v3,v6) 4
(v4,v7) 4
(v5,v7) 6
(v4,v5) 7
(v4,v6) 8
(v2,v5) 10
Kruskal’s Algorithm

\[(v1,v4) \quad 1 \quad OK\]
\[(v6,v7) \quad 1 \quad OK\]
\[(v1,v2) \quad 2 \quad OK\]
\[(v3,v4) \quad 2 \quad OK\]
\[(v2,v4) \quad 3 \quad reject\]
\[(v1,v3) \quad 4 \quad reject\]
\[(v3,v6) \quad 4 \quad \]
\[(v4,v7) \quad 4 \quad \]
\[(v5,v7) \quad 6 \quad \]
\[(v4,v5) \quad 7 \quad \]
\[(v4,v6) \quad 8 \quad \]
\[(v2,v5) \quad 10 \quad \]
Kruskal’s Algorithm

\[ (v1,v4) \quad 1 \quad OK \]
\[ (v6,v7) \quad 1 \quad OK \]
\[ (v1,v2) \quad 2 \quad OK \]
\[ (v3,v4) \quad 2 \quad OK \]
\[ (v2,v4) \quad 3 \quad reject \]
\[ (v1,v3) \quad 4 \quad reject \]
\[ (v3,v6) \quad 4 \quad reject \]
\[ (v4,v7) \quad 4 \quad reject \]
\[ (v5,v7) \quad 6 \quad OK \]
\[ (v4,v5) \quad 7 \quad OK \]
\[ (v4,v6) \quad 8 \quad OK \]
\[ (v2,v5) \quad 10 \quad OK \]
Kruskal’s Algorithm

(v1,v4) 1 OK
(v6,v7) 1 OK
(v1,v2) 2 OK
(v3,v4) 2 OK
(v2,v4) 3 reject
(v1,v3) 4 reject
(v3,v6) 4 reject
(v4,v7) 4 OK
(v5,v7) 6
(v4,v5) 7
(v4,v6) 8
(v2,v5) 10
Kruskal’s Algorithm

\[
\begin{align*}
(v_1, v_4) & \quad 1 & \text{OK} \\
(v_6, v_7) & \quad 1 & \text{OK} \\
(v_1, v_2) & \quad 2 & \text{OK} \\
(v_3, v_4) & \quad 2 & \text{OK} \\
(v_2, v_4) & \quad 3 & \text{reject} \\
(v_1, v_3) & \quad 4 & \text{reject} \\
(v_3, v_6) & \quad 4 & \text{reject} \\
(v_4, v_7) & \quad 4 & \text{OK} \\
(v_5, v_7) & \quad 6 & \text{OK} \\
(v_4, v_5) & \quad 7 \\
(v_4, v_6) & \quad 8 \\
(v_2, v_5) & \quad 10
\end{align*}
\]
Implementing Kruskal’s Algorithm

• Try to add edges one-by-one in increasing order. Build a heap in $O(|E|)$. Each deleteMin takes $O(\log |E|)$

• How to maintain the forest?

  • Represent each tree in the forest as a set.
  
  • When adding an edge, check if both vertices are in the same set. If not, take the union of the two sets.

  • This can be done efficiently using a disjoint set data structure (Weiss Chapter 8).

Total turns out to be: $O(|E| \log |V|)$
Application: Hierarchical Clustering

- This is a very common data analysis problem.
- Group together data items based on similarity (defined over some feature set).
- Discover classes and class relationships.
Zoo Data Set

101 animals represent each data item as a vector of integers (15 attributes).

<table>
<thead>
<tr>
<th></th>
<th>bear</th>
<th>chicke</th>
<th>tortoise</th>
<th>flea</th>
</tr>
</thead>
<tbody>
<tr>
<td>hair</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>feathers</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>eggs</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>milk</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>airborne</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>aquatic</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>predator</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>toothed</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>backbone</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>breathes</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>venomou</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>fins</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>legs</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>tail</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>domestic</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Zoo Data Set MST

- MST over 12 random animals.
Zoo Data Set MST

- Remove k-1 lowest cost edges to produce k clusters.
Zoo Data Set MST

- Remove k-1 lowest cost edges to produce k clusters.
Zoo Data Set MST

- Remove k-1 lowest cost edges to produce k clusters.
Zoo Data Set MST

- Remove \( k-1 \) lowest cost edges to produce \( k \) clusters.