• All Shortest Paths

• Questions from exercises and exams

---

All Shortest Paths

• The Problem: $G = (V, E, w)$ is a weighted directed graph. We want to find the shortest path between any pair of vertices in $G$.
• Example: find the distance between cities on a road map.
• Can you use already known algorithms?

---

All Shortest Paths

• From every vertex in the graph Run
  – Dijkstra: $O(|V||E|\log|V|) = O(|V|^2\log|V|)$
  – Run Bellman-Ford: $O(|V|^2|E|) = O(|V|^3)$

• Can we do better?
Dynamic Programming

• Dynamic Programming is a technique for solving problems “bottom-up”:
  • first, solve small problems, and then use the solutions to solve larger problems.
• What kind of problems can Dynamic Programming solve efficiently?

• Optimal substructure: The optimal solution contains optimal solutions to sub-problems.
• What other algorithms can suit this kind of problems?
  • Greedy algorithms
• Overlapping sub-problems: the number of different sub-problems is small, and a recursive algorithm might solve the same sub-problem a few times.

All Shortest Paths

• How can we define the size of sub-problems for the all shortest paths problem? (two way)
• Suggestion 1: according to the maximal number of edges participating in the shortest path (what algorithm uses this idea?)
• Suggestion 2: according to the set of vertices participating in the shortest paths (Floyd-Warshall)
All Shortest Paths - Suggestion 1

- The algorithm uses the $|V| \times |V|$ matrix representation of a graph.
- The result matrix - cell $(j,k)$ contains the weight of the shortest path between vertex $j$ and vertex $k$.
- Initialization: paths with 0 edges. What actual values are used?
  - $d_{i,k} = \infty$ for $i \neq k$, $d_{i,i} = 0$
- In iteration $m$, we find the shortest paths between all vertices with no more than $m$ edges and keep them in the matrix $D^{(m)}$. How many iterations are needed?

All Shortest Paths - Suggestion 1

- No circles with negative weights - $|V| - 1$ iterations.
- In iteration $m$:
  - For every $(v,u)$, find the minimum of:
    - The current shortest path $v \rightarrow u$ (maximum $m-1$ edges).
    - For every $w$ in $\text{Adj}(u)$: The shortest path with maximum $m$ edges through $w$, which is the shortest path $v \rightarrow w$ with maximum $m-1$ edges, plus the edge $(w,u)$.

All Shortest Paths - Suggestion 1

- Time complexity:
  - $|V|$ iterations
  - In each iteration: going over $O(|V|^2)$ pairs of vertices in
  - For each pair $(u,v)$: going over $O(|V|)$ possible neighbors
  - Total: $O(|V|^4)$
All Shortest Paths - Suggestion 1

- Improvement: If we know the shortest paths up to \( m \) edges long between every pair of vertices, we can find the shortest paths up to \( 2m \) edges in one iteration:
- For \((v, u)\) - the minimal path through vertex \( w \) is \( v \rightarrow w \rightarrow u \), when \( v \rightarrow w \) and \( w \rightarrow u \) have at most \( m \) edges.
- Time complexity: \( O(|V|^3 \log |V|) \)

All Shortest Paths - Suggestion 1

- Can we use this method to solve single-source-shortest-paths?
- Yes - we can update only the row vector that matches the single source, by using the results of previous iterations and the weights matrix.
- Note that this version is similar to Bellman-Ford.

Floyd-Warshall Algorithm

- Intermediate vertices on path \( p = \langle v_p, \ldots, v_f \rangle \) are all the vertices on \( p \) except the source \( v_p \) and the destination \( v_f \).
- If we already know the all shortest paths whose intermediate vertices belong to the set \{1,\ldots,k-1\}, how can we find all shortest paths with intermediate vertices \{1,\ldots,k\}?
- Consider the shortest path \( p \) between \((i, j)\), whose intermediate vertices belong to \{1,\ldots,k\}
Floyd-Warshall Algorithm

• If k is not an intermediate vertex in p, then p is the path found in the previous iteration.
• If k is in p, then we can write p as i→k→j, where the intermediate vertices in i→k and k→j belong to \{1,...,k-1\}.
• The algorithm:
  – Initialize: \( D^{(0)} = W \)
  – For \( k = 1, ..., |V| \)
    • For \( i = 1, ..., |V| \)
      • For \( j = 1, ..., |V| \)
        \[ d^{(k)}_{i,j} = \min(d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}) \]
• Time complexity: \( O(|V|^3) \)

Johnson’s Algorithm

• We already wrote, debugged and developed emotional attachment to the Dijkstra and Bellman-Ford algorithms. How can we use them to efficiently find all-shortest-paths?
• Step 1: What should we do to successfully run Dijkstra if we are sure that there are no circles with negative weights?
• Johnson’s Algorithm
  • We can find a mapping from the graph’s weights to non-negative weights.
  • The graph with the new weights must have the same shortest paths.
• Step 2: How can we be sure that there are no negative weighted circles?
  • Simply run Bellman-Ford
Johnson’s Algorithm

- The algorithm:
  - Add a dummy vertex, \( v \), and an edge with weight 0 from \( v \) to every vertex in the graph.
  - The modified graph has the same negative circles.

- Run Bellman-Ford from \( v \) to find negative circles, if any.
- Use the shortest paths from \( v \) to define non-negative weights:
  - \( w'(s, t) = w(s, t) + h(s) - h(t) \)
  - Is \( W' \) non-negative?
  - Yes, due to the fact that \( h(t) \leq w(s, t) + h(s) \)

- Do shortest paths remain shortest?
  - Let \( p \) be a shortest path between \( v_0 \) and \( v_l \), then \( w'(p) = \sum w'(v_i, v_{i+1}) = \sum [w(v_i, v_{i+1}) + h(v_{i+1}) - h(v_i)] = w(p) + h(v_0) - h(v_l) \)
  - The term \( h(v_0) - h(v_l) \) is common to all paths between \( v_0 \) and \( v_l \), so the minimal \( w'(p) \) matches the minimal \( w(p) \)
Johnson’s Algorithm

- So - now we can use \( W' \) to run Dijkstra from each vertex in \( G \).
- Time complexity: \( O(VE + |V|^2 \log |V|) \)
- Good for sparse graphs

Questions From Previous exams

a) Define Spanning Tree and Minimal Spanning Tree.

Spanning Tree: Given a graph \( G=(V,E) \), a spanning tree \( T \) of \( G \) is a connected graph \( T=(V,E') \) with no cycles (same vertices, a subset of the edges).

For example, this graph has three spanning trees:
\{ (a,b),(a,c) \}, \{ (a,b),(b,c) \}, \{ (a,c),(b,c) \}

Minimal Spanning Tree (MST): Given a weighted graph \( G=(V,E,w) \), define the weight of a spanning tree \( T \) as \( w(T) = \sum_{e \in E} w(e) \). Then a minimal spanning tree \( T \) is a spanning tree with minimal weight, i.e. \( T \) satisfies:

\[ w(T) = \min \{ w(T') | T' \text{ is a spanning tree} \} \]

For example, this graph has two minimal spanning trees:
\{ (a,b),(b,c) \}, \{ (a,c),(b,c) \}
b) Either prove or disprove the following claim:

In a weighted (connected) graph, if every edge has a different weight then G has exactly one MST.

First notice that if the edge weights are not distinct, then the claim is incorrect, for example the previous graph.

- So, can we come up with a counter-example when weights are distinct? (no, but thinking about it for a few minutes sometimes helps...)

A useful feature of spanning trees

Claim: Suppose \( T_1 \) and \( T_2 \) are two spanning trees of \( G \). Then for any edge \( e_i \) in \( T_1 \setminus T_2 \), there exists an edge \( e_j \) in \( T_2 \setminus T_1 \) such that \( T \setminus \{ e_i \} \cup \{ e_j \} \) is also a spanning tree.

To see this, consider the following partition of \( G \):

```
\begin{align*}
&\text{G}_u \quad \text{e}_1 \\
&\text{v} \quad \text{u'} \\
&\text{G}_v \quad \text{e}_2 \\
&\text{v'} \quad \text{u} \\
\end{align*}
```

Proof: Suppose \( e_i = (v,u) \). Denote by \( G_v \) and \( G_u \) the two connected components of \( G \) when removing \( e_i \) from \( T_1 \).

Examine the path from \( v \) to \( u \) in \( T_1 \); there must be an edge \( e_j = (v',u') \) in \( T_1 \) such that \( v' \) is in \( G_v \) and \( u' \) is in \( G_u \).

Let \( T'' = T \setminus \{ e_i \} \cup \{ e_j \} \)

- \( T'' \) is connected and has no cycles, thus it is a spanning tree, as claimed.

Take two vertices \( x \) and \( y \) in \( G \). If both are in \( G_v \) or in \( G_u \), then there is exactly one path from \( x \) to \( y \) since \( G_v \) and \( G_u \) are connected with no cycles. If \( x \) is in \( G_v \) and \( y \) is in \( G_u \), then there is also exactly one path between them: from \( x \) to \( v' \), then to \( u' \), and then to \( y \).
Back to the Question

Claim: In a weighted (connected) graph, if every edge has a different weight, then G has exactly one MST.

Proof: Suppose by contradiction that there are two MSTs, \( T_1 \) and \( T_2 \). Suppose also that the largest edge in \( T_1 \setminus T_2 \) is larger than the largest edge in \( T_2 \setminus T_1 \) (notice they can’t be equal). Let \( e_1 \) be the largest edge in \( T_1 \setminus T_2 \). There is an edge \( e_2 \) in \( T_2 \setminus T_1 \) such that \( T' = T_1 \setminus \{e_1\} \cup \{e_2\} \) is a spanning tree with weight:

\[
w(T') = w(T_1) + [w(e_2) - w(e_1)] < w(T_1)
\]

so \( T_1 \) is not an MST -> Contradiction.

Wrong proof for this claim

• A common (but wrong) argument from exams: “The Generic-MST algorithm always has a unique safe edge to add, thus it can create only one MST.”

• Why this is wrong?
  – There might be other ways to find an MST besides the Generic-MST algorithm.
  – It is not true that there is always one unique safe edge (!) For example, Prim and Kruskal might choose a different edge at the first step, although they are both Generic-MST variants

Questions From Previous Exams

c) Write an algorithm that receives an undirected graph \( G=(V,E) \) and a sub-graph \( T=(V,E_T) \) and determines if \( T \) is a spanning tree of \( G \) (not necessarily minimal).

• What do we have to check?
  • Cycles - run DFS on \( T \) and look for back edges
  • Connectivity - if there are no cycles, it is enough to check that \( |E_T| = |V| - 1 \).
Question 2

a) Both in Dijkstra and in Prim we have a set of nodes $S$ (that initially contains only $s$), and we add one additional node in each iteration. Prove or disprove that in both algorithms the nodes are added to $S$ in the same order.

The claim is not correct.

A contradictory example:

- Prim takes $s, a, b, c$
- Dijkstra takes $s, a, c, b$


Question 2 - difficult

b) Consider a directed graph with positive weights. Give an algorithm that receives a node $s$ and prints the shortest cycle that contains $s$.

- Suggestion 1: for every outgoing edge from $s$, $(s, v)$, find the shortest path from $v$ to $s$.
- Suggestion 2: Add a new node $s'$, and for every edge $(s, v)$ add an edge $(s', v)$ with the same weight. Now find a shortest path from $s'$ to $s$.


Question 3

- An in-order tree walk can be implemented by finding the minimum element and then making $n-1$ calls to TREE–SUCCESSOR
- How many times at most do we pass through each edge?
Question 3

TREE-SUCCESSOR(x)
if x.right==null         going up (1)
y=x.parent
while y!=null &&
x=x.right             going up (2)
y=y.parent
else
  y=x.right
  while y.left!=null
    y=y.left
return y

Question 3

• Right edges:
  - A right edge n→n.right is passed downwards only at (3), which happens when we call TREE-SUCCESSOR(n).
  - Since we call TREE-SUCCESSOR once for each node, we go down each right edge once, at most
• Left edges:
  - After we pass a left edge n→n.left (at (1) or (2)), TREE-SUCCESSOR returns n.
  - Since TREE-SUCCESSOR returns each node once, we go up each left edge once, at most
• Therefore, we pass each edge at most twice
• In-order walk takes O(n) steps

Question 4

• You are in a square maze of n×n cells and you’ve got loads of coins in your pocket. How do you get out?
• The maze is a graph where
  – Each cell is a node
  – Each passage between cells is an edge
• Solve the maze by running DFS until the exit is found
DFS - Reminder

<table>
<thead>
<tr>
<th>DFS(G)</th>
<th>DFS-VISIT(u)</th>
</tr>
</thead>
<tbody>
<tr>
<td>for each ( u \in V[G] )</td>
<td>u.color=gray</td>
</tr>
<tr>
<td>u.color=white</td>
<td>u.d=++time</td>
</tr>
<tr>
<td>u.prev=nil</td>
<td>for each ( v \in \text{adj}[u] )</td>
</tr>
<tr>
<td>time=0</td>
<td>if ( v).color=white</td>
</tr>
<tr>
<td>for each ( u \in V[G] )</td>
<td>v.prev=u</td>
</tr>
<tr>
<td>if u.color=white</td>
<td>DFS-VISIT(v)</td>
</tr>
<tr>
<td>DFS-VISIT(u)</td>
<td>u.color=black</td>
</tr>
<tr>
<td>u.f=++time</td>
<td></td>
</tr>
</tbody>
</table>

Question 4

- What does each color represent in the maze?
  - White - a cell without any coins
  - Gray - a cell with a coin lying with its head side up
  - Black - a cell with a coin lying with its tail side up

- An edge connecting a node to its parent is marked by a coin.
- When visiting a cell, we color it gray.
- If it has a white cell adjacent to it – visit it.
- If there are no such cells,
  - Color the cell “black” by flipping the coin.
  - backtrack by going to the cell marked as parent.

Question 4

- Each node has one parent.
- When backtracking, the parent will be the only adjacent “gray” cell that has a coin leading to it.

- Can we solve it using BFS?
  - No! In DFS we go between adjacent cells; in BFS, the nodes are in a queue, so the next cell could be anywhere.