Tirgul 14

- 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: $O(|V||E|\log|V|) = O(|V|^3\log|V|)$
  - Run Bellman-Ford: $O(|V|^2|E|) = O(|V|^4)$
- 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?

Dynamic Programming

- 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 $|V|x|V|$ 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 then 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* ~> *u* (maximum m-1 edges)
    - For every *w* in Adj*(u):* The shortest path with maximum m edges trough *w*, which is the shortest path *v*~>*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 2*m* edges in one iteration:
- For *(v,u)* - the minimal path through vertex *w* is *v*~>*w*~>*u*, when *v*~>*w* and *w*~>*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 $p = \langle v_1,\ldots,v_l \rangle$ are all the vertices on *p* except the source v_1 and the destination v_l.
- If we already know the all shortest paths whose intermediate vertices belong to the set {1,…,k-1}, how can we find all shortest paths with intermediate vertices {1,…,k}?
- Consider the shortest path *p* between (*i, j*), whose intermediate vertices belong to {1,…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* $k = 1 \ldots |V|$
    - *For* $i = 1 \ldots |V|$
      - *For* $j = 1 \ldots |V|$
        - » $d^{(k)}_{i,j} = \min(d_{i,j}^{(k-1)}, d_{ik}^{(k-1)} + d_{k,j}^{(k-1)})$
- Time complexity: $O(|V|^3)$

Johnson's Algorithm

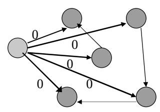
- We already wrote, debugged and developed emotional attachement to the Dijkstra and Bellman-Ford algorithms. How can we use them to efficiently find all-shortest-paths?
- Step 1: What should we do to successfully run Dijkstra if we are sure that there are no circles with negative weights?

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.

Johnson's Algorithm

- 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 $h(t) \leq w(s,t) + h(s)$

Johnson's Algorithm

- Do shortest paths remain shortest?
- Let p be a shortest path between v_0 and v_l, then $w'(p) = \Sigma w'(v_{i-1}, v_i) = \Sigma[w(v_{i-1}, v_i) + h(v_{i-1}) - h(v_i)] = w(p) + h(v_0) - h(v_l)$
- 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: $O(VE + |V|^2|E|\ log|V|)$
- Good for sparse graphs

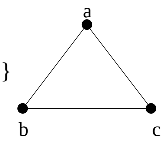
Questions From Previous exams

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)}



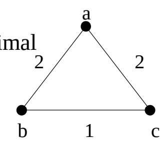
Questions From Previous exams

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

$$w(T) = \min\{w(T') \mid T' \text{ is a spanning tree}\}$$

For example, this graph has two minimal spanning trees:

{(a,b);(b,c)}, {(a,c);(b,c)}



Questions From Previous exams

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.

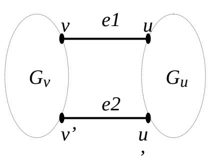
First notice that if the edge weights are not distinct, then the claim is incorrect, for example the previous graph.

- 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_1 in $T_1 \backslash T_2$ there exists an edge e_2 in $T_2 \backslash T_1$ such that $T1 \backslash \{e1\} \cup \{e2\}$ is also a spanning tree.

To see this, consider the following partition of G:



∎

A useful feature of spanning trees

Proof: Suppose $e_1 = (v,u)$. Denote by G_v and G_u the two connected components of *G* when removing e_1 from T_1.

Examine the path from *v* to *u* in T_2: there must be an edge $e_2=(v',u')$ in T_2 such that *v'* is in G_v and *u'* is in G_u.

Let. $T' = T_1 \backslash \{e_1\} \cup \{e_2\}$

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 (<u>but wrong</u>) 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).

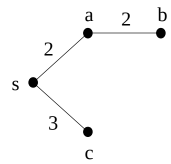
- What do we have to check?
- Cycles - run DFS on T and look for back edges
- Connectivity - if there are no cycles, it is enough to check that $|E_T|=|V|-1$.

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

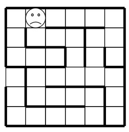
```
TREE-SUCCESSOR(x)
if x.right==null          <---- going up (1)
  y=x.parent
  while y!=null &&
  x==y.right              <---- going up (2)
     x=y
     y=y.parent           <---- going down (3)
else
  y=x.right               <---- going down (4)
  while y.left!=null
     y=y.left
return y
```

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 $n \times n$ 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

DFS(G)	DFS-VISIT(u)

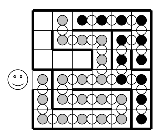
for each u∈ V[G]
  u.color=white
  u.prev=nil
time=0
for each u∈ V[G]
  if u.color=white
    DFS-VISIT(u)
	u.color=gray
	u.d=++time
	for each v∈ adj[u]
	  if v.color=white
	    v.prev=u
	    DFS-VISIT(v)
	u.color=black
	u.f=++time

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