GB21802 - Programming Challenges

Week 6 - Graph Problems (Part II)

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2015-06-03,6

Last updated June 3, 2016

Last Week Results

Week 5 - Graph I

- Denominator – 17/31
- Knight War Grid – 10/31
- Wetlands of Florida – 8/31
- Battleships – 3/31
- Pick up Sticks – 1/31
- Place the Guards – 1/31
- Street Dominos – 0/31
- Dominos – 0/31
- Freckles – 5/31
- Artic Network – 0/31

  - 10 people solved: 0 problems;
  - 7 people solved: 1 problem;
  - 3 people solved: 2 problems;
  - 10 people solved: 3-4 problems;
  - 1 person solved: 7 problems!

Special Notes

Week 5 and 6 – Outline

Last Week - Graph I

- Graph Basics review: Concepts and Data Structure;
- Depth First Search and Breadth First Search;
- Problems you solve with DFS and BFS;
- Minimum Spanning Tree: Kruskal and Prim Algorithms;

Next Week - Graph II

- Single Sourse Shortest Path; (Friday)
- All Pairs Shortest Path; (Friday)
- Network Flow; (Monday)
- Special Graphs and Related Problems; (Monday)

Many variations in graph problems!

SSSP: Single Source Shortest Path

Problem Definition

Given a graph G and a source s, what are the shortest paths from s to a target node t?

Common solutions:

- On unweighted graphs, BFS is good enough, but it won't work correctly in weighted graphs;
- Djikstra Algorithm (usually $O((V+E)\log V)$)
- Bellman Ford's Algorithm $O(VE)$

Reminder: SSSP on unweighted graph (BFS)

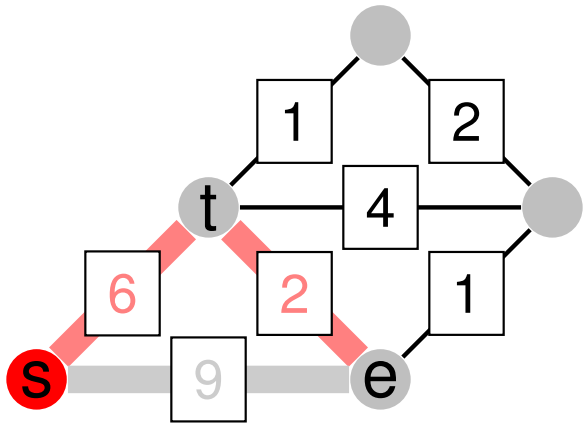
- Visit nodes from *s* to *t*;
- Calculate minimal cost from *s* to *t*;
- (optional) Print the path from *s* to *t*;

```
vector <int> p; // list of parents
void printPath(u) { // recursive print path s to u
   if (u == s) { cout << s; return; } // base case
   printPath(p[u]); cout << " " << u; }

vector <int> dist(V,INF); dist[s] = 0;
queue <int> q; q.push(s);
while (!q.empty()) {
   int u = q.front(); q.pop();
   for (int j = 0; j < (int)AdjList[u].size(); j++) {
      pair <int,int> v = AdjList[u][j];
      if (dist[v.first] == INF) {
         dist[v.first] = dist[u] + 1;
         p[v.first] = u; q.push(v.first); }}}
```

BFS: Problem with weighted graphs

Simple BFS will give a wrong answer when there is a longer path which is cheaper than a shorter one.



In this graph, BFS would find $s \to e$ (cost 9) as the shortest path from s to e, instead of $s \to t \to e$ (cost 8).

Djikstra's Algorithm for weighted SSSP

Basic Idea

Greedly selects the next edge that minimizes the total weight of the visited graph.

- Many different implementations (original paper has no implementation);
- A simple way is to use C++ stl's *Priority Queue*;
- When visiting a node, store new edges in the priority queue;
  (priority queue sort edges as they are inserted)
- For every new edge visited, check if new path on target node is cheaper than old one (if it is cheaper, visit the node again);

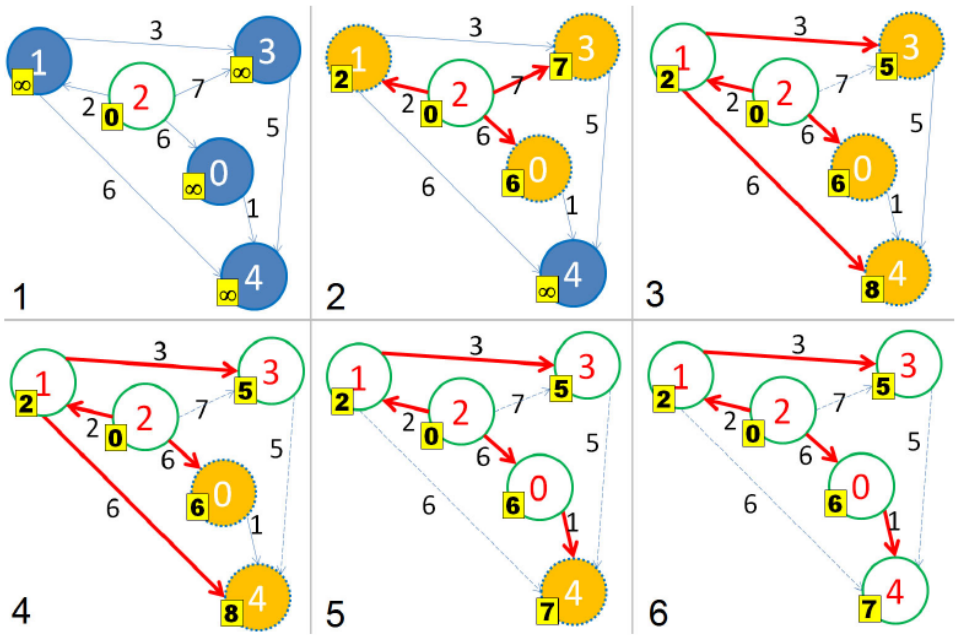
Djikstra's Algorithm: _A_ Implementation

```
Vector<int> dist(V,INF); dist[s] = 0;
priority_queue<ii, vector<ii>, greater<ii>> pq;
pq.push(ii,(0,s));
while (!pq.empty()) {
   ii front = pq.top(); pq.pop();
   // shortest unvisited vertex
   int d = front.first; u = front.second;
   if (d > dist[u]) continue; // *lazy deletion*
   for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[u] + v.second < dist[v.first]) {
         dist[v.first] = dist[u] + v.second // relax
         pq.push(ii(dist[v.first],v.first));}}}
         // new node to queue
```

This implementation uses *lazy deletion* to avoid deleting nodes from the queue unless absolutely necessary;

Djikstra's implementation trick: Lazy deletion

- Djikstra Requires us to store the edge to vertex v with cheapest weight in the queue;
- Removing an edge from the queue is expensive;
- Instead, we keep all edges, and test each visited edge;

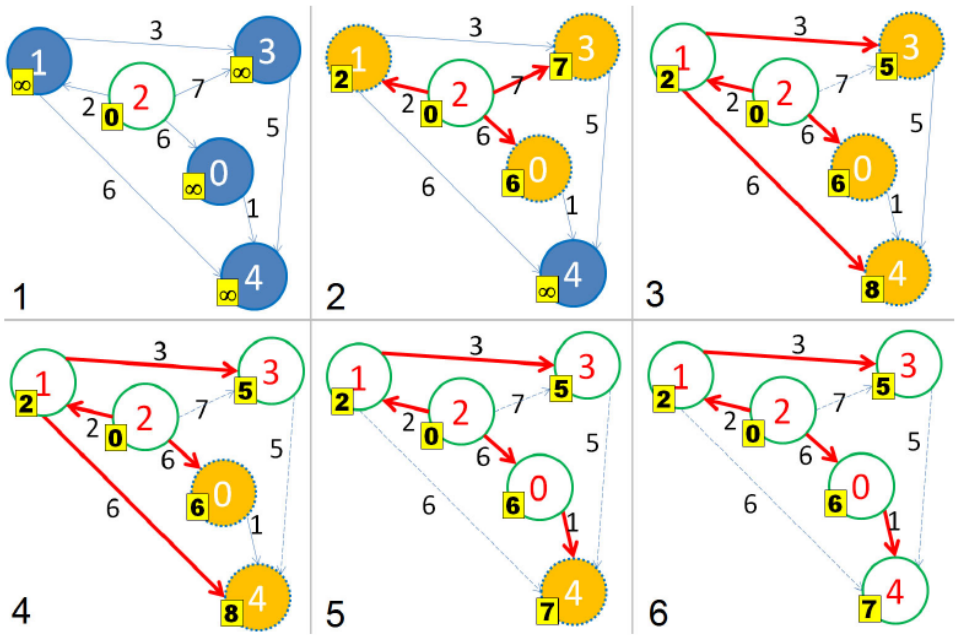


Vertex queue: [(2,1), (6,0),(7,3),];

Images from *Competitive Programming*, Steven Halim

Djikstra's implementation trick: Lazy deletion

- Djikstra Requires us to store the edge to vertex v with cheapest weight in the queue;
- Removing an edge from the queue is expensive;
- Instead, we keep all edges, and test each visited edge;

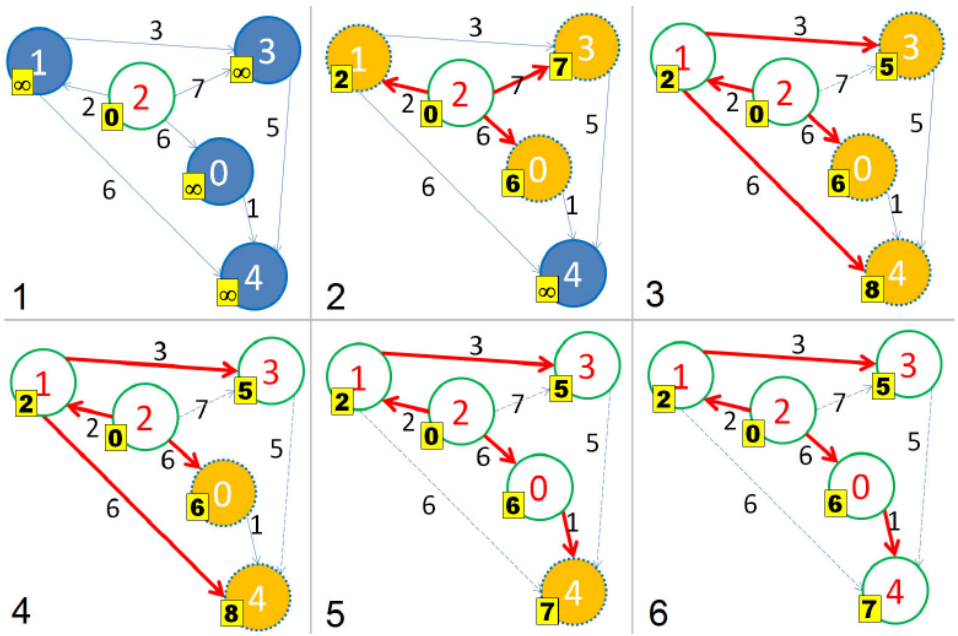


Vertex queue: [(5,3), (6,0),(7,3), (8,4),];

Images from *Competitive Programming*, Steven Halim

Djikstra's implementation trick: Lazy deletion

- Djikstra Requires us to store the edge to vertex v with cheapest weight in the queue;
- Removing an edge from the queue is expensive;
- Instead, we keep all edges, and test each visited edge;

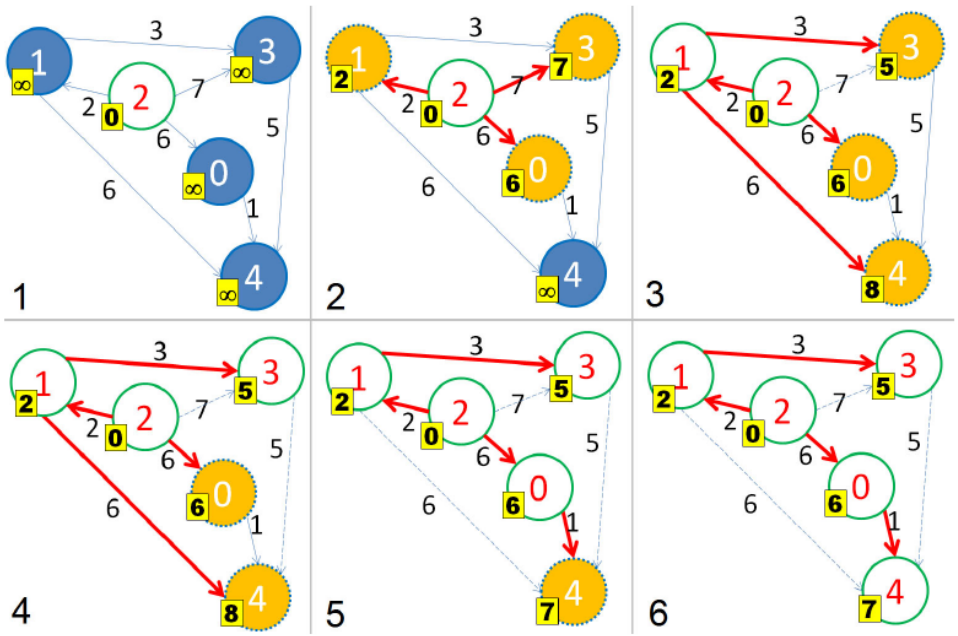


Vertex queue: [(6,0),(7,3), (8,4), (10,4)];

Images from *Competitive Programming*, Steven Halim

Djikstra's implementation trick: Lazy deletion

- Djikstra Requires us to store the edge to vertex v with cheapest weight in the queue;
- Removing an edge from the queue is expensive;
- Instead, we keep all edges, and test each visited edge;



Vertex queue: [(7,3), (7,4),(8,4), (10,4)];

Images from *Competitive Programming*, Steven Halim

Problem Example: UVA 11367 – Full Tank

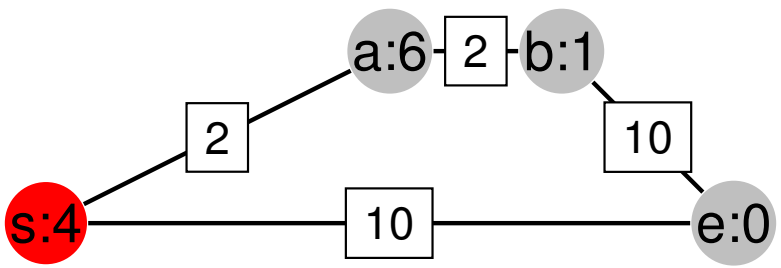
A big part of Graph problems is figuring out the correct graph representation.

Problem Summary

In a graph G of roads ($1 \leq V \leq 1000, 0 \leq E \leq 10000$) with the following information:

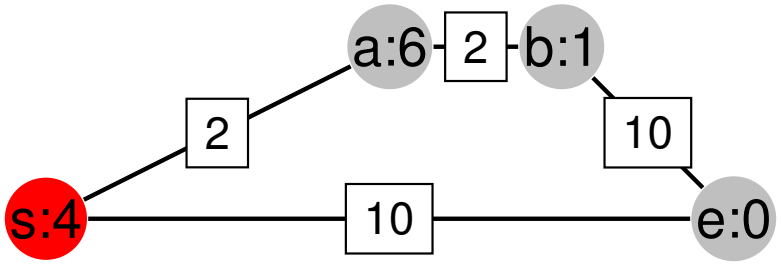
- cost l_{ij} in units of fuel between cities i and j;
- price p_i of buying fuel at city i;
- tank capacity c of a car;

What is the least fuel price from node s to e;



The regular shortest path is s to e, but the smallest fuel price is buying fuel at s (cost 4) and at a (cost 1), for a total cost 26

UVA 11367 – Full Tank – Problem modeling



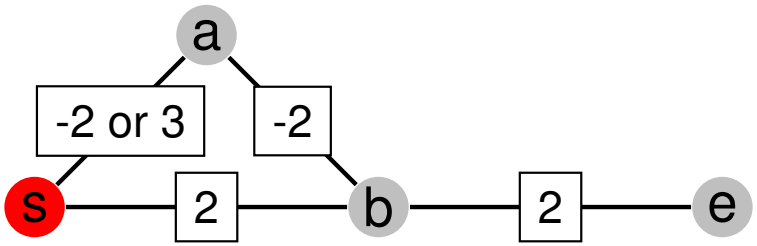
Simple Djikstra on the l_{ij} graph will not solve this problem, because we have to take into account the cost of buying fuel at different nodes.

To model this problem, we modify the graph:

- Transform nodes (i) into nodes (i, f) (how much fuel left we have at i);
- An edge $(i, f) \rightarrow (j, f - l_{ij})$ exists if $f - l_{ij} \geq 0$ and has cost 0 (spend existing fuel);
- An edge $(i, f) \rightarrow (i, f + 1)$ has cost p_i (buy fuel at the city)
- Use regular djikstra on the new graph;

Djikstra Problem with negative cycles

The djikstra implementation suggested earlier can deal with negative weights, but will enter an infinite loop if the graph has a negative loop.



- The Djikstra implementation above will keep adding negative nodes as smaller totals are generated: -2, -4, -6, -8...
- The Problem is that it does not include an explicit check for *non-simple* paths;
- To deal with negative loops, one alternative is the Bellman Ford's algorithm

Bellman Ford's Algorithm $O(VE)$

Main Idea: Relax all E edges in the graph $V - 1$ times;

```
vector<int> dist(V, INF); dist[s] = 0;
for (int i = 0; i < V - 1; i++) // repeat V-1 times
 for (int u = 0; u < V; u++) // for all vertices
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j]; // record path here if needed;
    dist[v.first] = min(dist[v.first], dist[u]+v.second);
    } //relax edge
```

How does it work?:

- At the start, *dist[s]* has the correct minimal distance;
- When relax all edges, at least one node now has correct minimal distance;
- After *V-1* iterations, all nodes have correct minimal distances;

A bit more on Bellman Ford's

Detecting Negative Loops

Bellman Ford's algorithm can be used to detect if a negative loop exists in the graph.

- Execute the algorithm once.
- Do one last round of "relax all nodes".
- If any distance in the *dist[]* vector changes, the graph has a negative loop.

Summary of SSSP

- BFS: $O(V + E)$, only for unweighted graphs
- Djikstra: $O(E + V \log V)$, problem with negative loops
- Bellman Ford: $O(EV)$, can find negative loops (simple to code)

Study/Implement all of them, but always use the simplest possible!

APSP: All Pairs Shortest Path

Problem Definition – UVA 11463 – Commandos

Given a graph $G(V, E)$, and two vertices s and e, calculate the smallest possible value of the path $s \to i \to e$ for every node $i \in V$.

- One way to do it would be for every node i, calculate Djikstra(s,i) + Djikstra(i,e);
- This would cost $O(V(E + V\log(V)))$;
- If the graph is small ($|V| \leq 400$), there is a simpler-to-code algorithm that costs $O(V^3)$

The Floyd-Warshall Algorithm – $O(V^3)$

```
int AdjMat[V][V];
//contains weight of edge(i,j) or INF if no such edge

for (int k=0; k < V; k++) // loop order is k -> i -> j
   for (int i=0; i < V; i++)
      for (int j=0; j < V; j++)
         AdjMat[i][j] = min(AdjMat[i][j],
                            AdjMat[i][k]+AdjMat[k][j]);
// AdjMat[i][j] now contains the minimal cost[i][j]
```

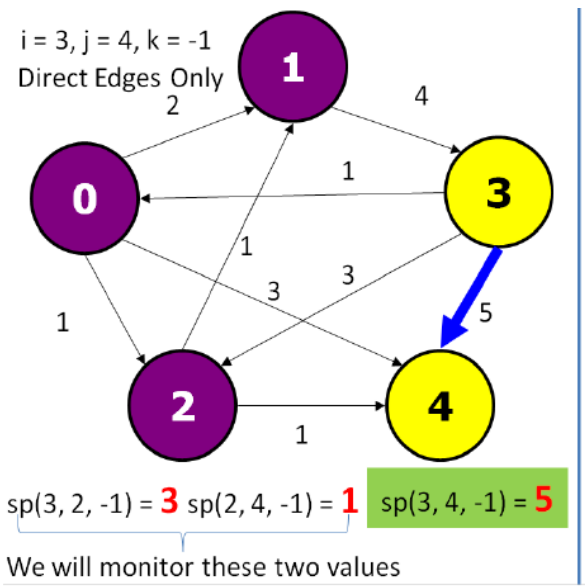
- Computational cost is more expensive than V times Djikstra;
- Programmer Cost is clearly cheaper than Djikstra or Bellman-Ford;

Why does Floyd Warshall work?

Basic Idea: Bottom-up Dynamic Programming

FW uses this recursive idea: "The shortest path S between i and j is either $S(i, j)$, or $S(i, v) + S(v, j)$ for all nodes v between 0 to k."

- $k = -1$ is the base case, S is the edge (i, j);
- For $k = n$, the shortest path uses $S(i, v, n - 1)$ and $S(v, j, n - 1)$;



The current content of Adjacency Matrix D at **k = -1**

k = -1	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	**5**
4	∞	∞	∞	∞	0

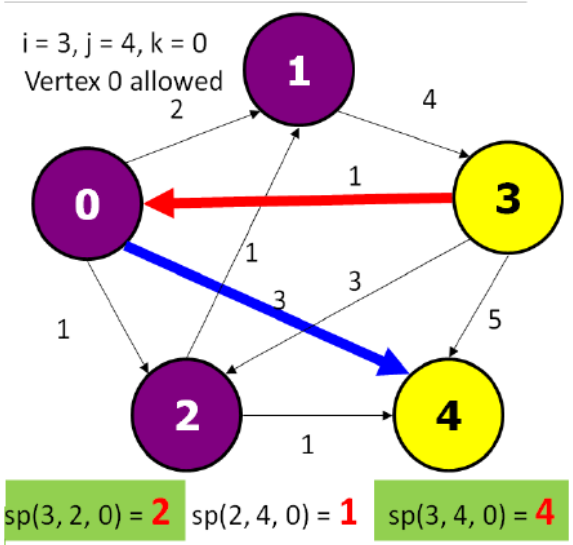
Images from *Competitive Programming*, Steven Halim

Why does Floyd Warshall work?

Basic Idea: Bottom-up Dynamic Programming

FW uses this recursive idea: "The shortest path S between i and j is either $S(i, j)$, or $S(i, v) + S(v, j)$ for all nodes v between 0 to k."

- $k = -1$ is the base case, S is the edge (i, j);
- For $k = n$, the shortest path uses $S(i, v, n - 1)$ and $S(v, j, n - 1)$;



The current content of Adjacency Matrix D at **k = 0**

k = 0	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

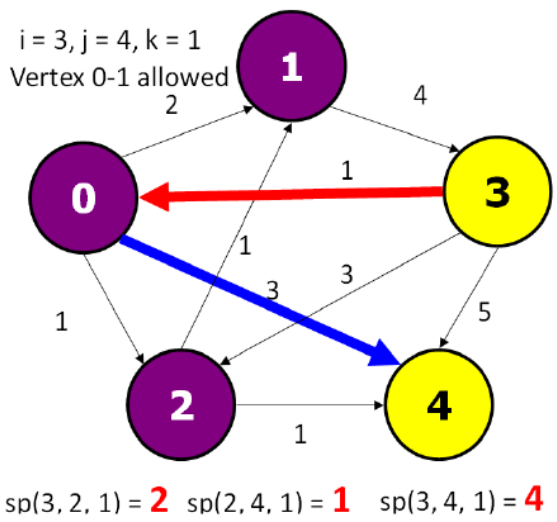
Images from *Competitive Programming*, Steven Halim

Why does Floyd Warshall work?

Basic Idea: Bottom-up Dynamic Programming

FW uses this recursive idea: "The shortest path S between i and j is either $S(i, j)$, or $S(i, v) + S(v, j)$ for all nodes v between 0 to k."

- $k = -1$ is the base case, S is the edge (i, j);
- For $k = n$, the shortest path uses $S(i, v, n - 1)$ and $S(v, j, n - 1)$;



The current content of Adjacency Matrix D at **k = 1**

k = 1	0	1	2	3	4
0	0	2	1	6	3
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	4
4	∞	∞	∞	∞	0

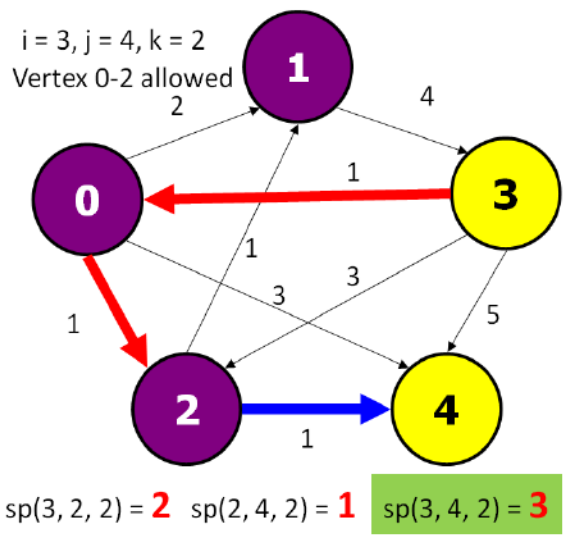
Images from *Competitive Programming*, Steven Halim

Why does Floyd Warshall work?

Basic Idea: Bottom-up Dynamic Programming

FW uses this recursive idea: "The shortest path S between i and j is either $S(i, j)$, or $S(i, v) + S(v, j)$ for all nodes v between 0 to k."

- $k = -1$ is the base case, S is the edge (i, j);
- For $k = n$, the shortest path uses $S(i, v, n - 1)$ and $S(v, j, n - 1)$;



The current content of Adjacency Matrix D at **k = 2**

k = 2	0	1	2	3	4
0	0	2	1	6	2
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	0

Images from *Competitive Programming*, Steven Halim

Tricks with ASPS

- Include parents: Add a 2D parent matrix, which is updated in the inner loop; Follow the parent matrix backwards when the execution is over;
- Connectivity: If all we want to know if is i is connected to j, do FW with bitwise operations (FW[i][j] = FW[i][k] && FW[k][j]) – much faster!
- Finding SCC: If FW[i][j] and FW[j][i] are both > 0, then i and j belong to the same SCC;
- Minimum Cycle/Negative Cycle: Check the diagonal of FW: $FW[i][i] < 0$;
- "Diameter" of a Graph: i, j where $FW[i][j]$ is maximum;

SSSP and ASPS problem discussion

- From Dusk Until Down;
- Wormholes;
- Mice and Maze;
- Degrees of Separation;
- Avoiding your Boss;
- Arbitrage;

Network Flow

Ford Fulkerson Method

One way to solve the Max Flow is the Ford Fulkerson Method
(Footnote: Same Ford as the Bellman-Ford algorithm)
Algorithm Idea:

```
// initial setup: residual graph - edge capacity =
mf =0
while ("a path p with positive weight exists" from
  find v (edge with minimal weight in p)
  remove Wv from all foward edges i in P(s,g)
  increase Wv to all backward edges i in P(g,s)
  mf += f
}
```

Question to think about: Why do we increase Wv to all backward edges?

Implementing Ford Fulkerson Method

“while (a path p with positive weight exists)“
How do we find this path? Examples: BFS, DFS. (Does not need to be minimal, the increase of Wv will fix that (but it may be slow))
Cost with DFS: O(|f*|E), where f* is the Max Flow value. But f* can be arbitrarily large!
Better idea: BFS (Edmond Karp’s algorithm)

Edmond Karp's algorithm implementation

Network Flow Problem Example: UVA 259

Network Flow Problem Variants

Graph Problems: Thinking with many boxes

The interesting part about graph problems is that they came in great variety. Once you know the basic algorithms, the main problem is figuring out what kind of graph you are dealing with. This knowlege only comes with practice, but here are some examples.

DAG example: Timed SSSP

Max Flow Example: Iceberg

Bipartite Example - Prime pairing

Flow and Special Graph problems

Summary

This Week's Problems

Next Weeks

- Week 7
- Week 8
- Week 9