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 7, 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)

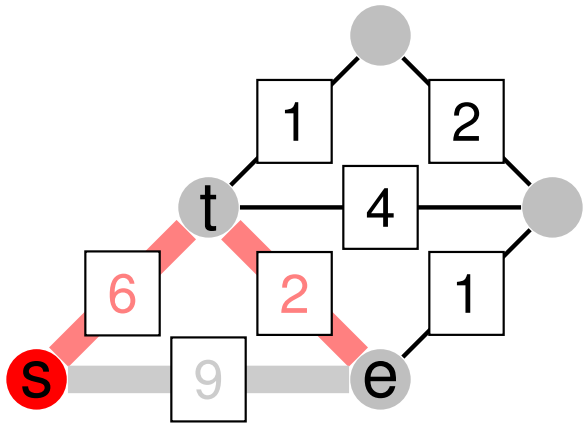
- Visit nodes from *s* to *t*;
- Calculate minimal cost from *s* to *t*;
- (optional) Print the path from *s* to *t*;

```
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);

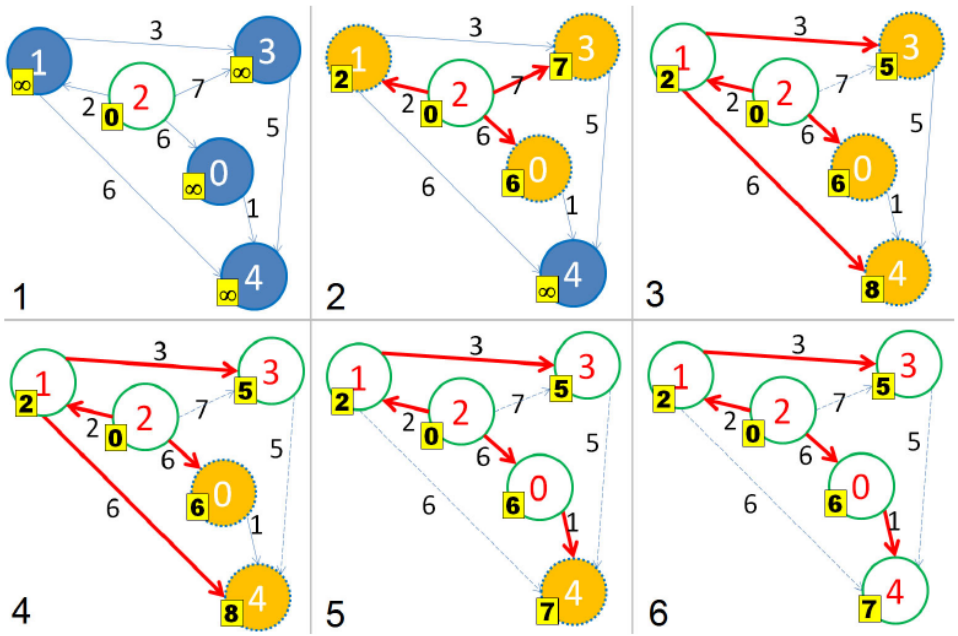
Djikstra's Algorithm: _A_ Implementation

```
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;

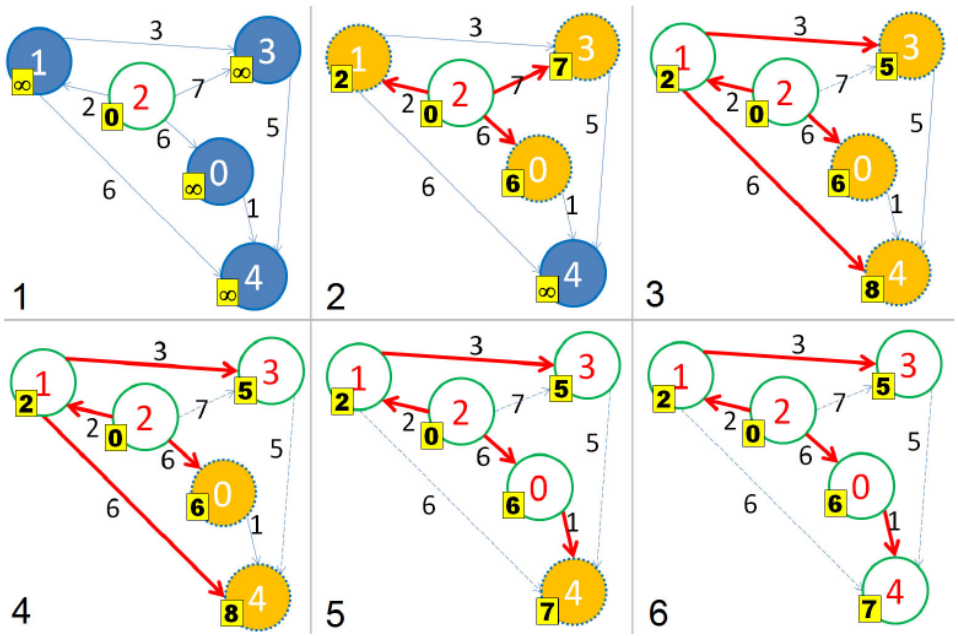


Vertex queue: [(2,1), (6,0),(7,3),];

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;

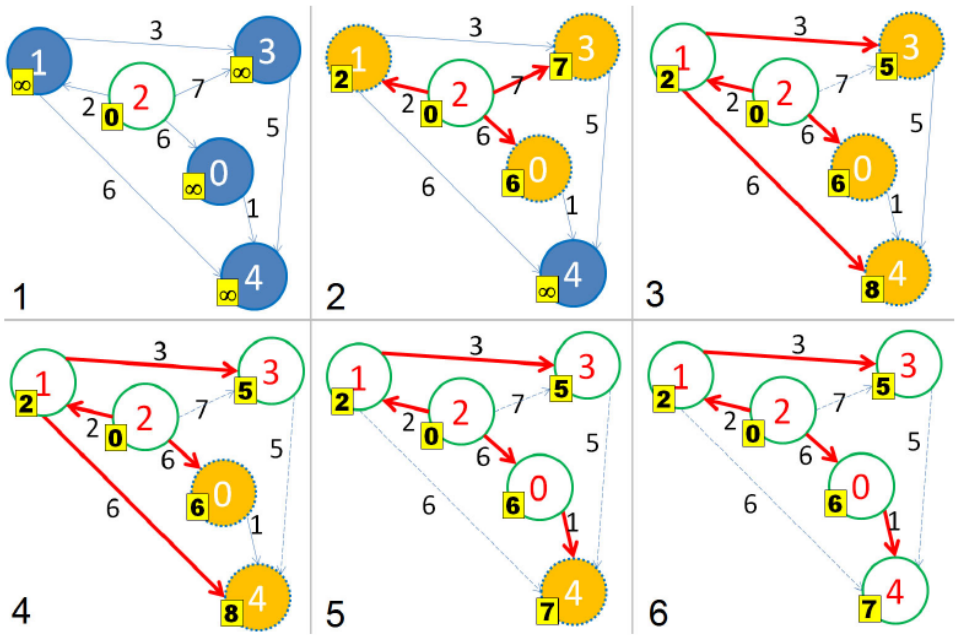


Vertex queue: [(5,3), (6,0),(7,3), (8,4),];

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;

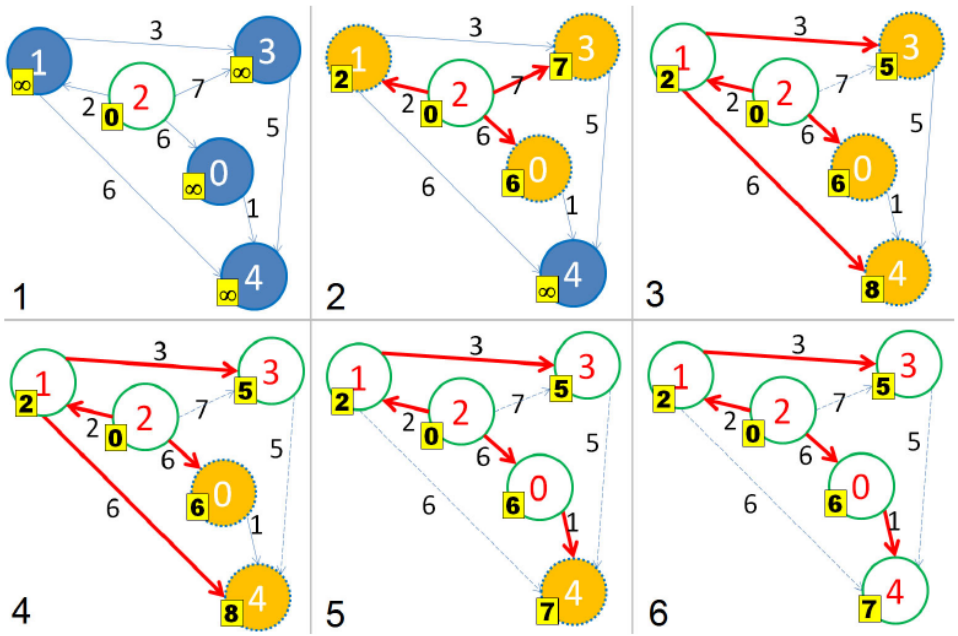


Vertex queue: [(6,0),(7,3), (8,4), (10,4)];

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

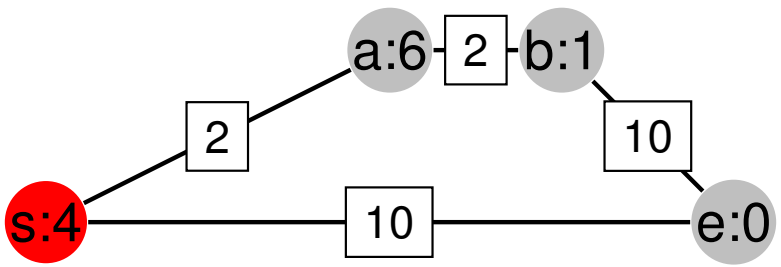
A big part of Graph problems is figuring out the correct graph representation.

Problem Summary

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

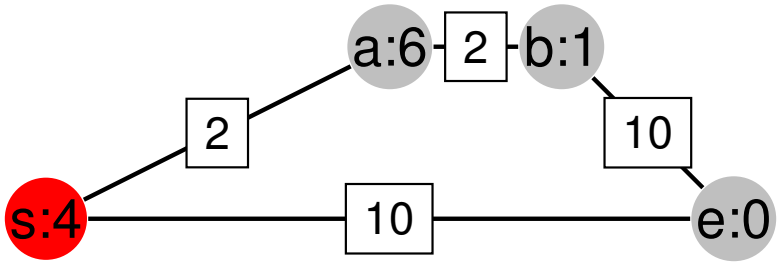
- cost l_{ij} in units of fuel between cities i and j;
- price p_i of buying fuel at city i;
- tank capacity c of a car;

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



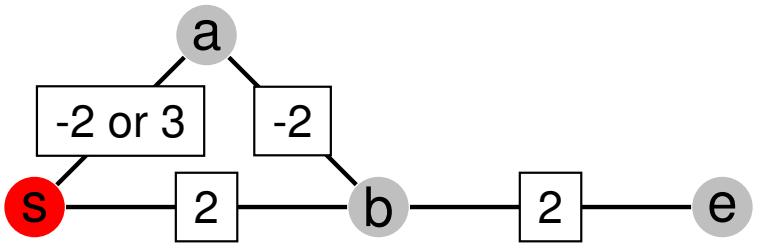
Simple Djikstra on the l_{ij} graph will not solve this problem, because we have to take into account the cost of buying fuel at different nodes.

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]
```

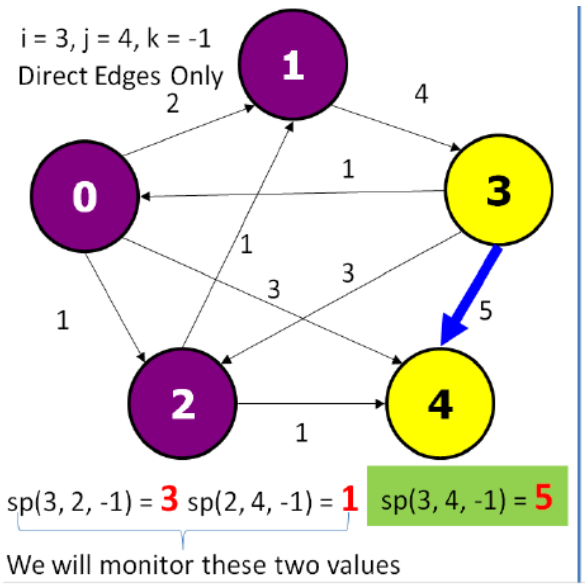
- Computational cost is more expensive than V times Djikstra;
- Programmer Cost is clearly cheaper than Djikstra or Bellman-Ford;

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

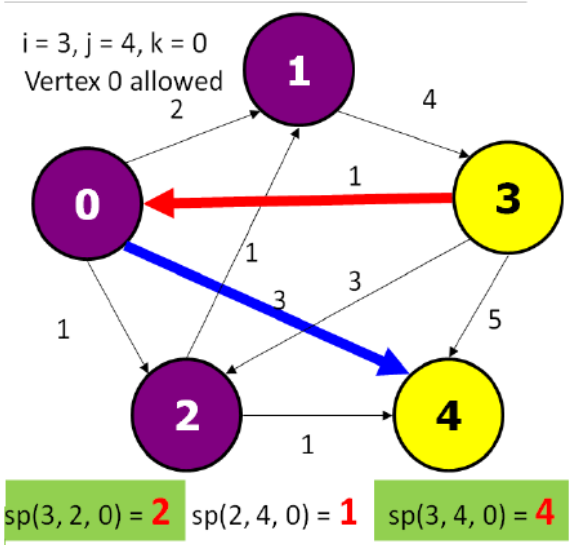
Images from *Competitive Programming*, Steven Halim

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

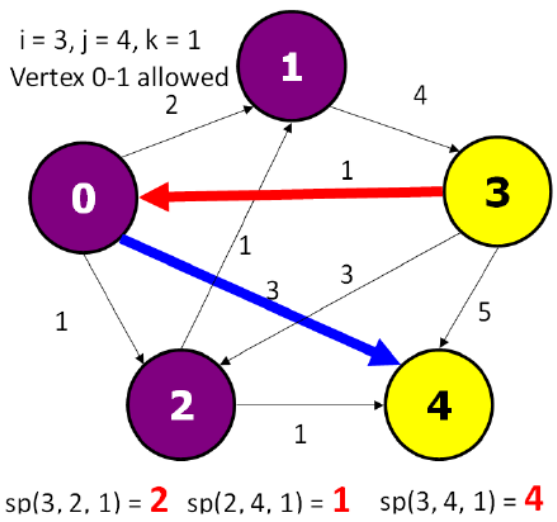
Images from *Competitive Programming*, Steven Halim

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

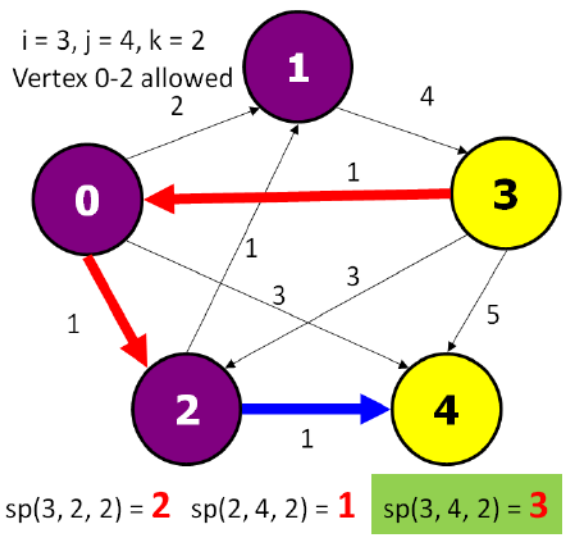
Images from *Competitive Programming*, Steven Halim

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;

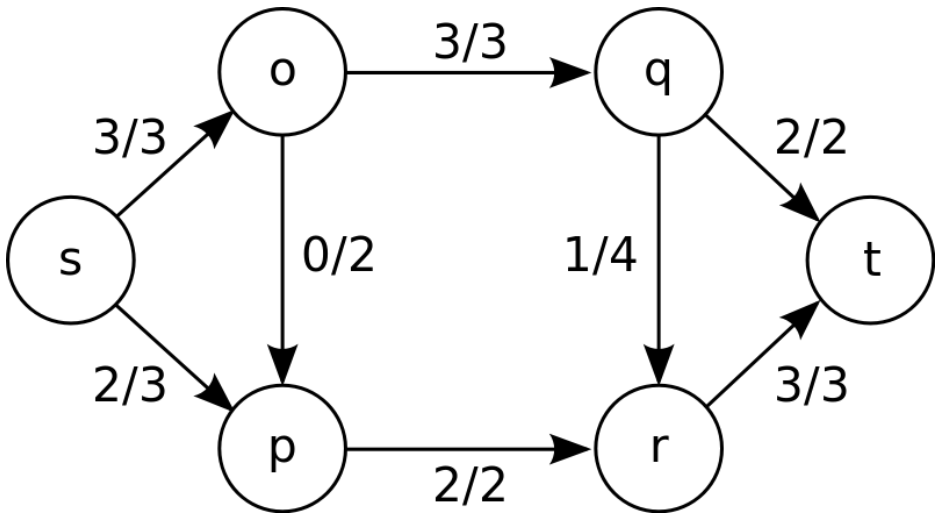
SSSP and ASPS problem discussion

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

Network Flow

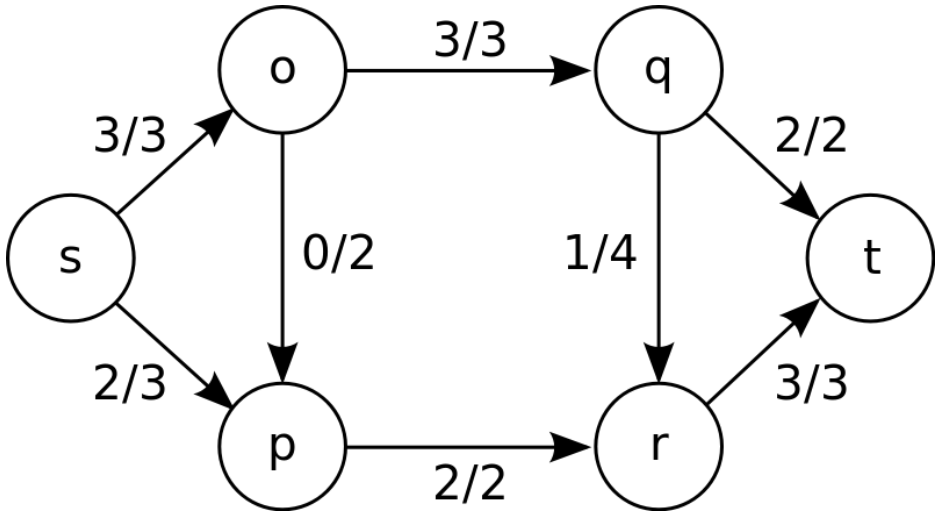
Problem Definition

Imagine a connected, weighted and directed network of pipes. From the source node, an infinite amount of water is entering. How much water is leaving at the destination node?



This image describes the MAX FLOW problem. It is part of a family of graph problems called "flow problems".

Network Flow



In the image above, 5 “flow” go from s to t.

- 2 flow go through: $s \to o \to q \to t$;
- 1 flow go through: $s \to o \to q \to r \to t$;
- 2 flow go through: $s \to p \to r \to t$;

Ford Fulkerson Method

One way to solve the Max Flow problem is using the Ford-Fulkerson Method.

Note: Same Ford as in Bellman-Ford

Basic Idea:

1. Create residual graph with flow capacity between nodes;
2. Initial residual graph = weight graph. Non existing edges have capacity 0;
3. Find a path between s and t;
4. Remove value of lowest weight from all edges in residual graph;
5. Add value of lowest weight to all reverse edges in residual graph;
6. goto 3;

Ford Fulkerson – Pseudocode

```
int residual[V][V];
memset(residual,0,sizeof(residual))
for (int i; i < AdjList.size();i++)
   for (int j; j < AdjList[i].size();j++)
      residual[i][AdjList[i][j].target] =
         AdjList[i][j].weight;

mf = 0;
while (P = FindPath(s,t)) > 0) {
   m = P.weight; // minimum edge in P
   for (v in P) {
      residual[v.first][v.second] -= m;
      residual[v.second][v.first] += m;}
   mf += m;
}
```

Ford Fulkerson Implementation: Finding Paths

```
while (P = FindPath(s,t) > 0) {...}
```

A key part of the algorithm is finding paths from s to t, and calculating the flow of the path (minimum edge weigth). We can use any algorithm to implement this search.

Idea 1: Find augmenting paths using DFS;

This is simple to implement, but the worst case scenario has cost: $O(|f * |E)$, where $f*$ is the true Max Flow value.

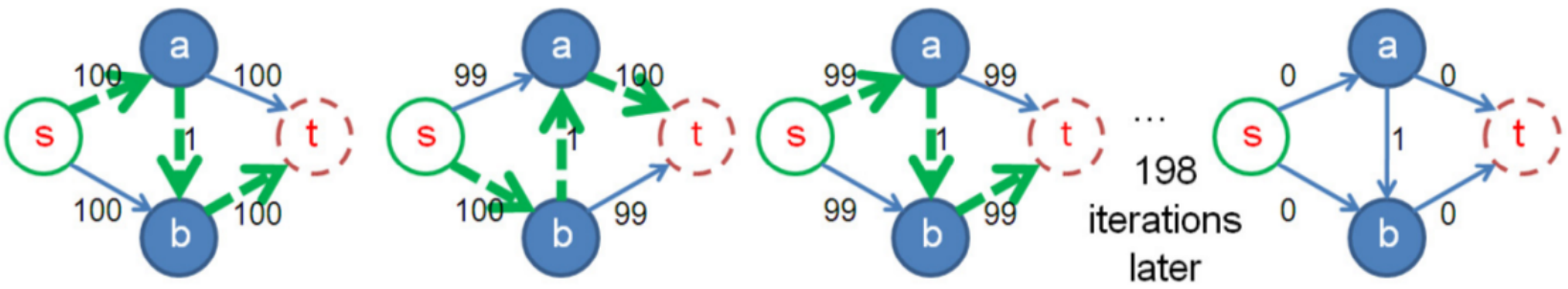


Image from Competitive Programming, 3rd edition

Edmond Karp's Algorithm

O(|f*|E) is too expensive!

The algorithm presented in the last page is too expensive for arbitrary values of |f*|. The Edmond Karp algorithm is a $O(VE^2)$ variation using BFS.

```
//// PART 1: Augment based on BFS

// This is a simpler O(V^3E) implementation
int res[MAX_V][MAX_V], mf, s, t;
vi p; // BFS spanning tree from s

void augment(int v, int minEdge) { // traverse BFS s->t
   inf (v == s) { f = minEdge; return; }
   else if (p[v] != -1) {
      augment(p[v], min(minEdge,res[p[v]][v]));
      res[p[v]][v] -= f; res[v][p[v]] += f;}}
```

Edmond Karp's Algorithm (2)

O(|f*|E) is too expensive!

The algorithm presented in the last page is too expensive for arbitrary values of |f*|. The Edmond Karp algorithm is a $O(VE^2)$ variation using BFS.

```
//// PART 2: create the BFS
mf = 0;
while (1) { f = 0; vi dist(MAX_V,INF): dist[s] = 0;
            queue<int> q; q.push(s); p.assign(MAX_V,-1);
   while (!q.empty()) { int u = q.front; q.pop();
      if (u==t) break; for int (v = 0; v < MAX_V; v++)
         if (res[u][v] > 0 && dist[v] == INF)
            dist[v] = dist[u]+1, q.push(v), p[v] = u; }
   // BFS (parent indexes) is now in "p";
   augment(t,INF); // modify residuals based on P
   if (f == 0) break; // if no more flow, end.
   mf += f;
   }
```

NF Example: UVA 259 – Software Allocation

Problem Description

- (up to) 26 programs and 10 computers;
- Each computer can run only one program at a time;
- Each computer has a list of programs it can run;
- Each program wants to run in "p" computers;

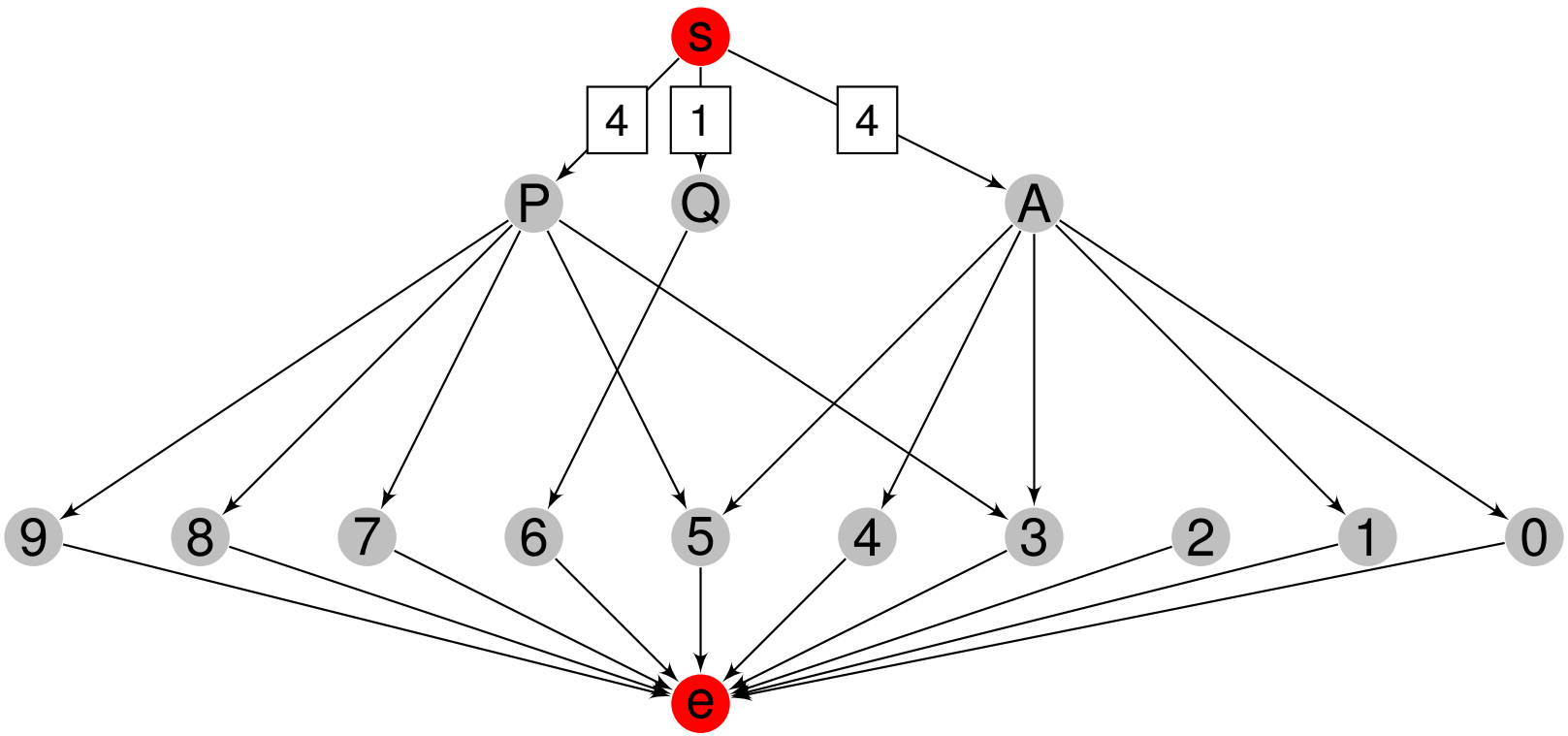
Can you find an allocation to run all programs?

Allocation Problems (Often called "matching" problems) can be solved using the Max Flow algorithm.

Create a source node connected to all the requesters; and an end node connected to all the providers. The max flow between the source and the end shows how much of the allocation is possible.

NF Example: UVA 259 – Software Allocation

```
A4 01345;
Q1 6
P4 35789;
```



Network Flow Problem Variants (1)

Minimum Cut – UVA 10480 (Sabotage)

Find a minimum set of edges C so that if all edges from the set are removed, then the flow from s to e becomes 0. (i.e., s and e are disconnected).

Ideas:

- Perform the Max Flow, and analyse the residual matrix;
- All nodes reachable from s using edges with positive residual are connected to s;
- All nodes not reachable from s as above are connected to e;
- All edges with residual 0 connect the two sets, and belong to the Minimum Cut;

Network Flow Problem Variants (2)

Multiple Sources, Multiple Sinks

Ideas:

- Create a “super source” node ss. ss connects to all sources with infinite weight;
- Create a “super sink” node se. All sinks connect to se with infinite weight;

Weights on nodes, not edges

Ideas:

- Split the vertices. Vertice weight is now an edge connecting both halves.
- Be careful. Doing this doubles V and increases E.
- (note how many times the solution is to modify the graph to our needs)

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: Fishmonger (SPOJ 0101)

Problem Description

You are given a number of cities $3 \leq n \leq 50$, remaining time $1 \leq t \leq 1000$, a toll matrix and a travel time matrix, choose a route:

- starting at city 0 and ending at city $n - 1$;
- travel time must be less than t;
- toll must be minimum;

There are two goals: *time* and *cost* that can contradict each other. How do you calculate the SSSP in this condition?

DAG example: Fishmonger (SPOJ 0101)

To handle the time constraint, we add a parameter (state) to each vertex indicating how much time is left.

The number of nodes will be multiplied:
$(1) \leftarrow (1, t), (1, t-1), (1, t-2), \ldots, (1, 0)$,
but now the graph is an DAG (time can only move in one direction).

But because it is a DAG, the problem becomes solvable using DP search (*find(city,timeleft)*).

Network Flow Example 2: Titanic (UVA 11380)

Problem Description

You are given a map of an accident:

```
# -- large wood: safe place, houses P people;
@ -- large iceberg: unsafe, houses 1 person;
. -- ice: unsafe, 1 person, breaks after using once;
~ -- freezing water: unsafe, no one can use;
* -- initial position of each person;
```

How many people can find paths to the safe place?

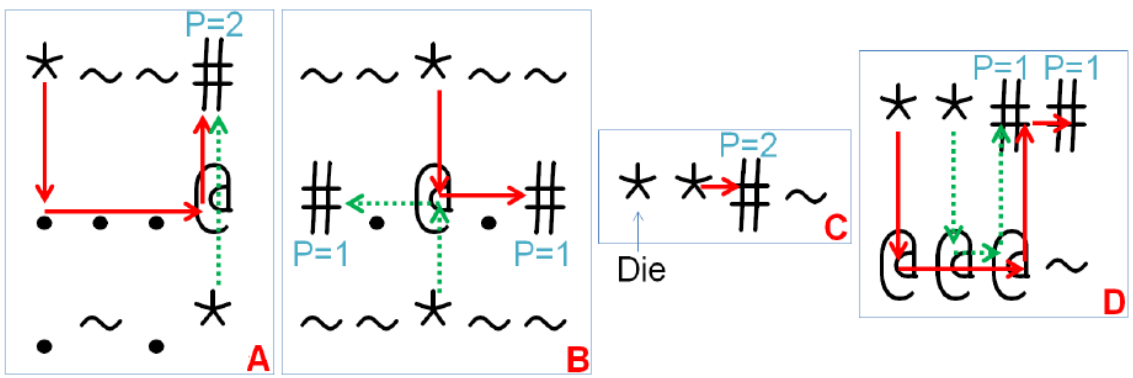


Image from *Competitive Programming, 3rd edition*

Network Flow Example 2: Titanic (UVA11380)

Graph Modeling

Since we have many people trying to go to the safe places, we can model this problem as a Max Flow problem. How do we set up the graph?

Network Flow Example 2: Titanic (UVA11380)

Graph Modeling

Since we have many people trying to go to the safe places, we can model this problem as a Max Flow problem. How do we set up the graph?

- Connect all non “water” cells with INF capacity to represent the paths;
- Set vertex capacity: ice and initial position is 1, iceberg and large wood is INF, to represent breaking;
- Super source node connect to all survivors with capacity 1;
- Super sink node connect to all large woods with capacity P;

Bipartite Example - Prime pairing

Problem Description

Two numbers a, b can be prime paired if their sum $a + b$ is prime. Given a set of numbers N, print the pairings of $N[0]$ that allow a complete pairing of N to be made.

Examples:

- $N = \{1, 4, 7, 10, 11, 12\}$ – answer: $\{4, 10\}$

Is this even a graph problem??

Bipartite Example - Prime pairing

The trick is to note that this is an “allocation” problem, just like “software allocation”. Each number will be allocated to another number to form primes.

How to create the graph:

- Split set between odds and evens (because odd + odd is not prime);
- Create edges between odds and evens that sum primes;
- super source connects to odds, super sink connects to edges;
- see if the max flow is equal to the number of nodes/2;

Summary

- BFS for unweighted path search; Djikstra for weighted path search;
- Bellman-ford for weighted path search with negative loops;
- Floyd-Warshall for all-to-all path search;
- Ford-Fergusson for Network flow

- However, the most important skill in graph problems is knowing how to transform the problem into a graph that can be solved by a known algorithm;

This Week's Problems

- From Dusk Until Down;
- Wormholes;
- Mice and Maze;
- Degrees of Separation;
- Avoiding your Boss;
- Arbitrage;
- Software Allocation;
- Sabotage;
- Little Red Riding Hood;
- Gopher II;

Next Weeks

- Week 7 – Math Problems;
- Week 8 – Computational Geometry;
- Week 9 – String Problems;

Hang in there a little bit more! :-)