GB21802 - Programming Challenges

Week 5 - Graph Problems (Part II)

Claus Aranha
caranha@cs.tsukuba.ac.jp

College of Information Science

2019-05-24,27

Last updated May 23, 2019

Graphs II – Outline

Algorithms

- Single Sourse Shortest Path;
- All Pairs Shortest Path;
- Network Flow;

Also, we will see a few variations on these algorithms.

SSSP: Single Source Shortest Path

Problem Definition

I want to go from vertex s (source) to t (target). What is the path with the **minimum sum of edge weights**?

Common solutions:

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

Reminder: SSSP on unweighted graph (BFS)

- Start in *s*, visit all nodes
- Save cost and parent node in each visited node;

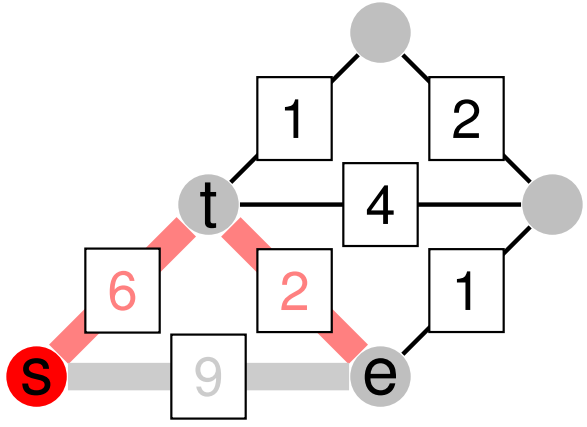
```
vector <int> p;                              // parent list

vector <int> dist(V,INF); dist[s] = 0;// initialize
queue  <int> q;                 q.push(s);
while (!q.empty()) {
   int u = q.front(); q.pop();
   for (int j = 0; j < AdjList[u].size(); j++) {
      int v = AdjList[u][j];
      if (dist[v] == INF) {                  // not visited
         dist[v] = dist[u] + 1;
         p[v] = u; q.push(v); }}}

void printPath(u) {                          // path from (u)
   if (u == s) { cout << s; return; }
   printPath(p[u]); cout << " " << u; }
```

BFS: Problem with weighted graphs

Just BFS is fast, but if the graph has weights, you may get a WRONG ANSWER.



- BFS shortest path: $s \to e$ (cost 9)
- Real shortest path: $s \to t \to e$ (cost 8)

Dijkstra's Algorithm for weighted SSSP

Basic Idea: Greedy Algorithm

Always add the edge with the shortest weight to the next node.

- 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;
- However! Deleting nodes in a priority queue is expensive!
  - We gain a bit of time by "skipping" longer paths (trick!)

Dijkstra's Algorithm: One Implementation Example

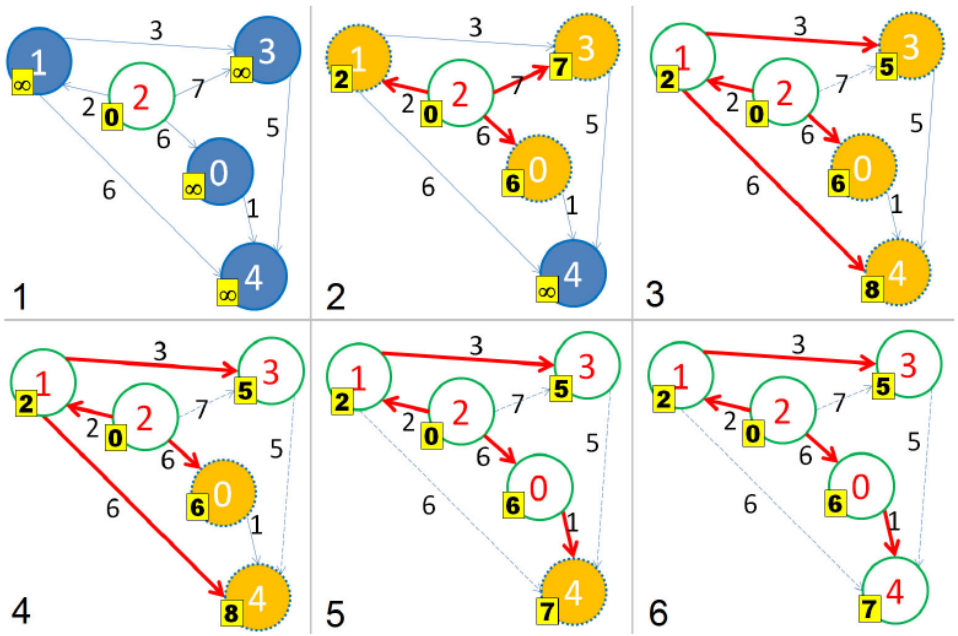
```
typedef pair<int,int> ii;                    // <node, weight>
priority_queue<ii, vector<ii>, greater<ii>> pq;

Vector<int> dist(V,INF); dist[s] = 0;    // initial dist
pq.push(ii,(0,s));

while (!pq.empty()) {
   ii front = pq.top(); pq.pop();          // first unvisited
   int d = front.first; u = front.second;
   if (d > dist[u]) continue;              // *lazy deletion*
   for (int j = 0; j < AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                // <node, weight>
      if (dist[u] + v.second < dist[v.first]) {
                                           // update node
         dist[v.first] = dist[u] + v.second;
         pq.push(ii(dist[v.first],v.first));
}}}
```

Dijkstra's implementation trick: Lazy deletion

- Dijkstra 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;

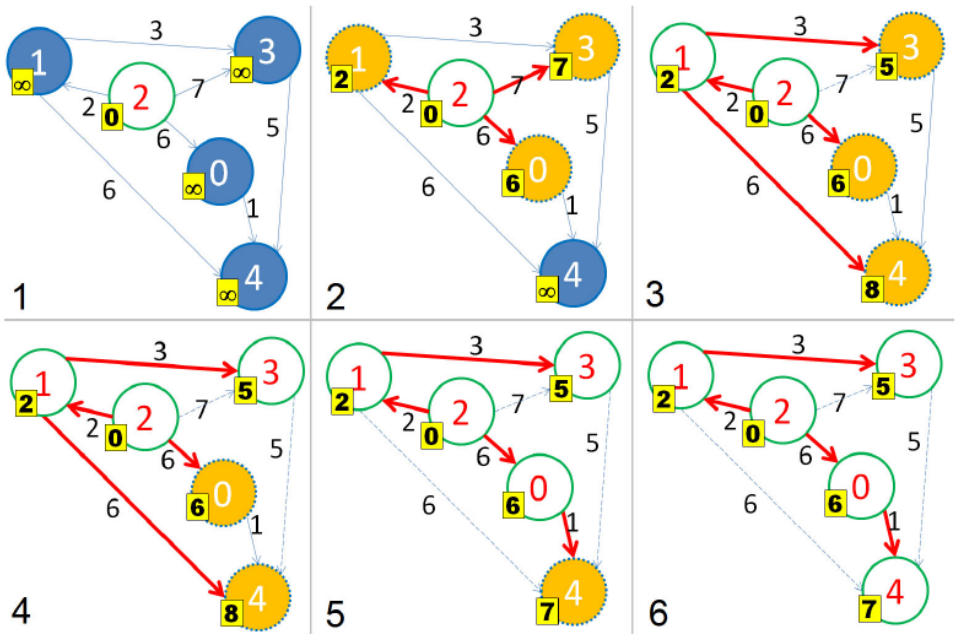


Vertex queue: [(2,1), (6,0),(7,3),];

Images from *Competitive Programming*, Steven Halim

Dijkstra's implementation trick: Lazy deletion

- Dijkstra 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;

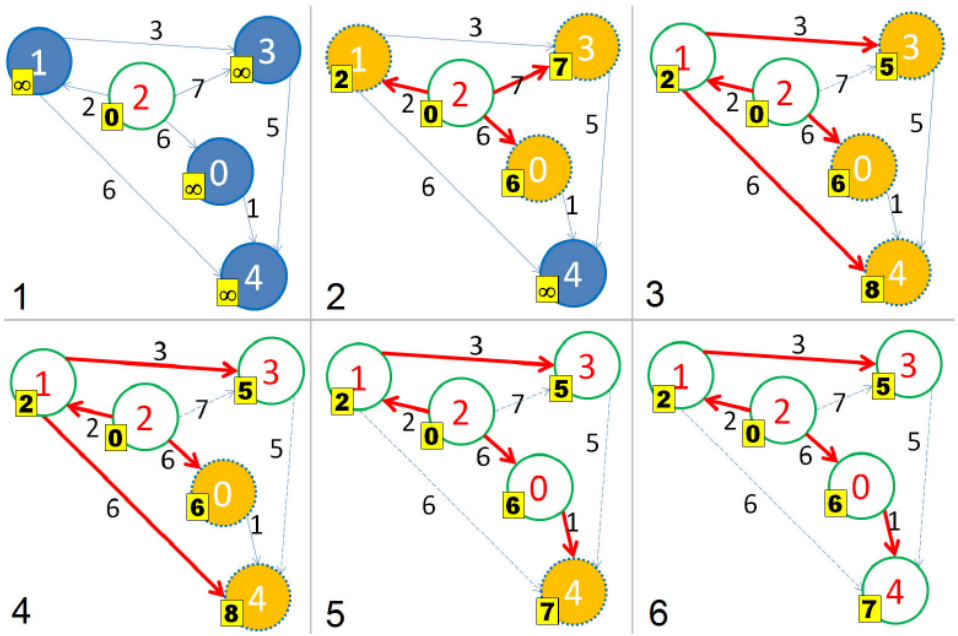


Vertex queue: [(5,3), (6,0),(7,3), (8,4),];

Images from *Competitive Programming*, Steven Halim

Dijkstra's implementation trick: Lazy deletion

- Dijkstra 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;

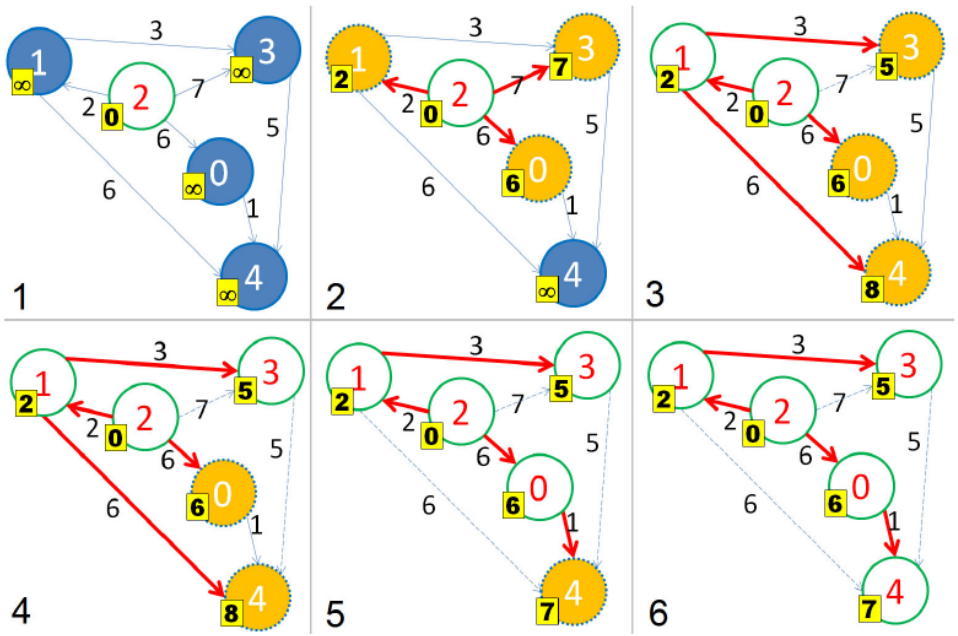


Vertex queue: [(6,0),(7,3), (8,4), (10,4)];

Images from *Competitive Programming*, Steven Halim

Dijkstra's implementation trick: Lazy deletion

- Dijkstra 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

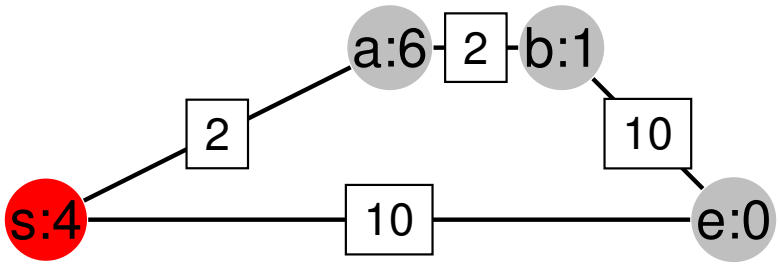
Problem Example: UVA 11367 – Full Tank

Problem Summary

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

- It takes F_{ij} units of fuel to go from i to j;
- You can buy one unit of fuel at i for the price p_i;
- The car has maximum capacity c;

What is the **minimum** price to go from s to e?

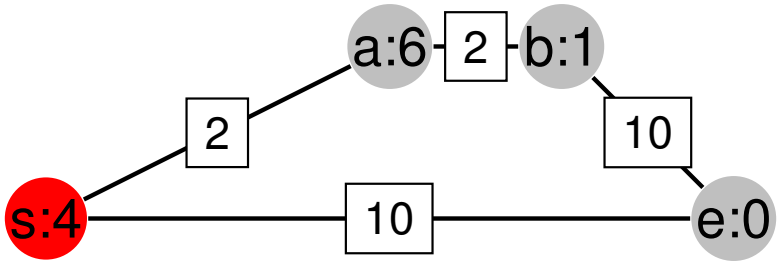


If you go straight from s to e, you only use 10 fuel, but the fuel is expensive!

If you go $s \to a \to b \to e$ you pay less for the fuel.

UVA 11367 – Full Tank – Problem modeling

Ok, but how do we implement this solution?



Using Dijkstra directly is hard – how do we model buying gas for multiple vertices? One idea is to **modify the graph**

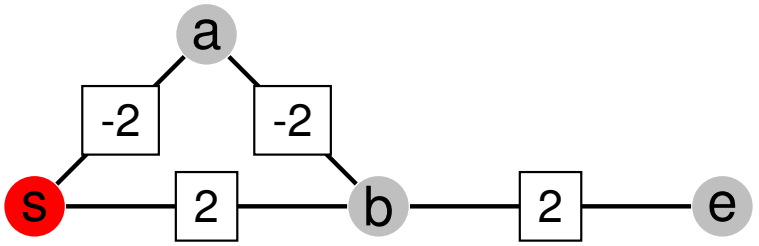
- Transform nodes (i) into nodes (i, f) (how much fuel left we have at i);
- An edge $(i, f) \to (j, f - F_{ij})$ exists if $f - F_{ij} \geq 0$. This edge has cost 0;
- An edge $(i, f) \to (i, f + 1)$ has cost p_i (buy one unit of fuel at node i)

We can use djikstra on this new graph, but it has many more nodes!

A Problem with Dijkstra

What happens if our graph has negative weights?

The djikstra implementation we discussed works with **negative weights**, but not with **negative loops**!



- Our initial Djikstra implementation will find repeated smaller costs from *s* to *a*: -2, -4, -6, -8...
- It needs a check to avoid **using the same edge two times**
- One simple way to solve this problem is using the Bellman Ford's algorithm

Bellman Ford's Algorithm $O(VE)$

Main Idea: update all $i \leftarrow j$ weights $V - 1$ times;

Version using Edge List

```
vector<int> dist(V, INF); dist[s] = 0;// Start Condition
int edges[E][3];                      // Edge list (i,j,w)
for (int i = 0; i < V - 1; i++)       // repeat V-1 times
 for (int u = 0; u < E; u++) {        // for all edges
  dist[edges[u][1]] = min(dist[edges[u][1],
                          dist[edges[u][0]+edges[u][2]);
 }
```

How does it work?:

- At the start, *dist[s]* has the correct minimal distance;
- When we update all edges, one new node now has correct distance;
- After $V - 1$ updates, all nodes have correct minimal distance;

Bellman Ford's Algorithm $O(VE)$

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

Version using Adjacency Matrix – Cost $O(V^3)$

```
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
```

- This version is one order of magnitude more expensive!
- But depending on the input, maybe it is difficult to get an edge list;
- Think about the input size before choosing!

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: $O(EV)$ or $O(V^3)$, can find negative loops

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 $O(V(E + V\log(V)))$;
- There is a simpler algorithm that costs $O(V^3)$. You can use it if the graph is *small*

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]
```

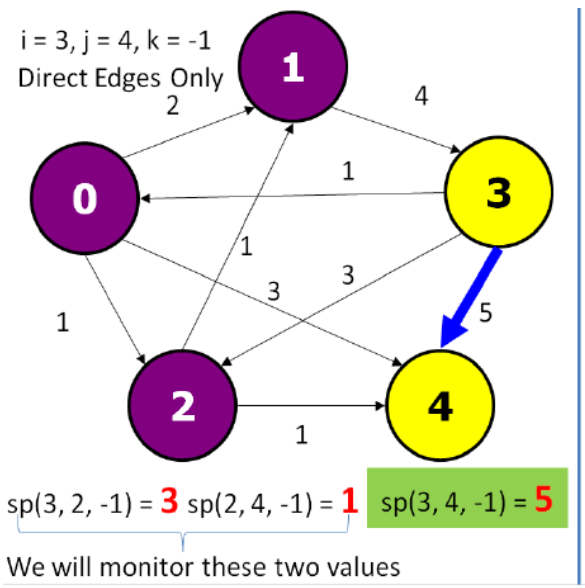
- Computational cost is more expensive than V times Djikstra;
- Programmer Cost is clearly cheaper than Djikstra or Bellman-Ford;

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 the minimum of $S(i, j)$ or $(S(i, v) + S(v, j))$ for all nodes v between 0 and 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

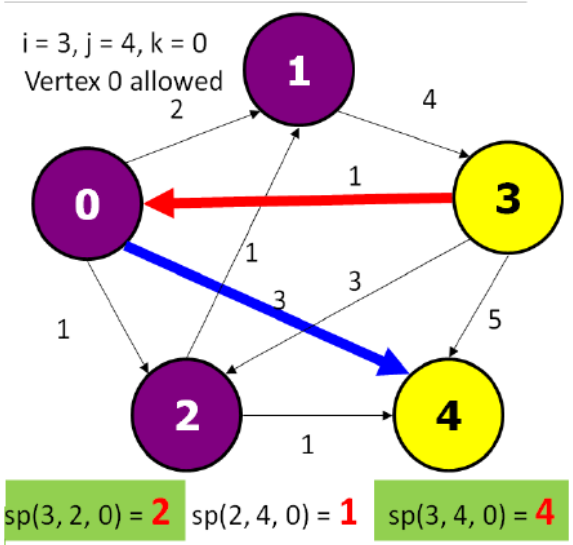
Images from *Competitive Programming*, Steven Halim

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 the minimum of $S(i, j)$ or $(S(i, v) + S(v, j))$ for all nodes v between 0 and 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

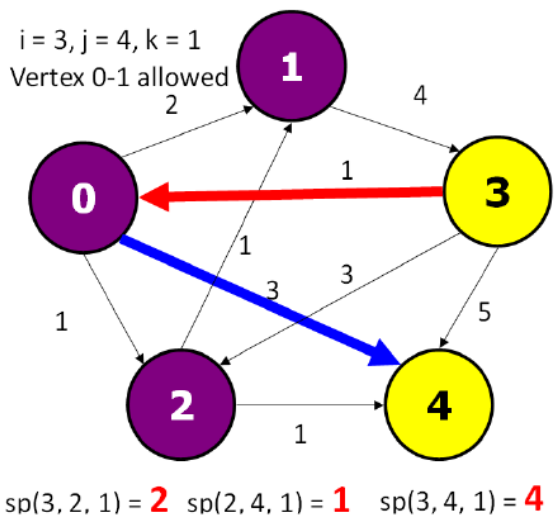
Images from *Competitive Programming*, Steven Halim

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 the minimum of $S(i, j)$ or $(S(i, v) + S(v, j))$ for all nodes v between 0 and 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

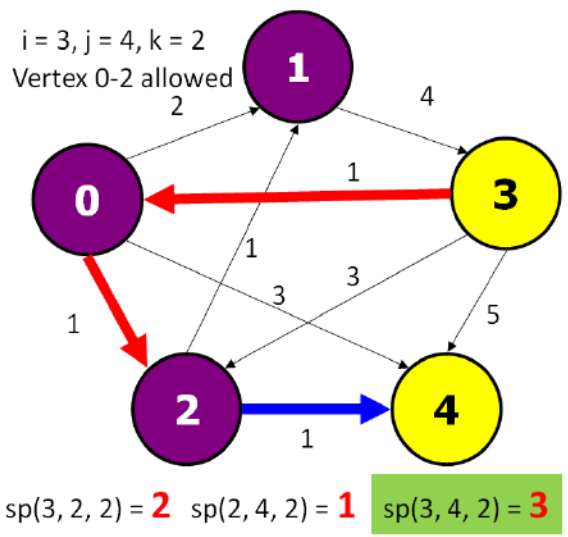
Images from *Competitive Programming*, Steven Halim

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 the minimum of $S(i, j)$ or $(S(i, v) + S(v, j))$ for all nodes v between 0 and 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

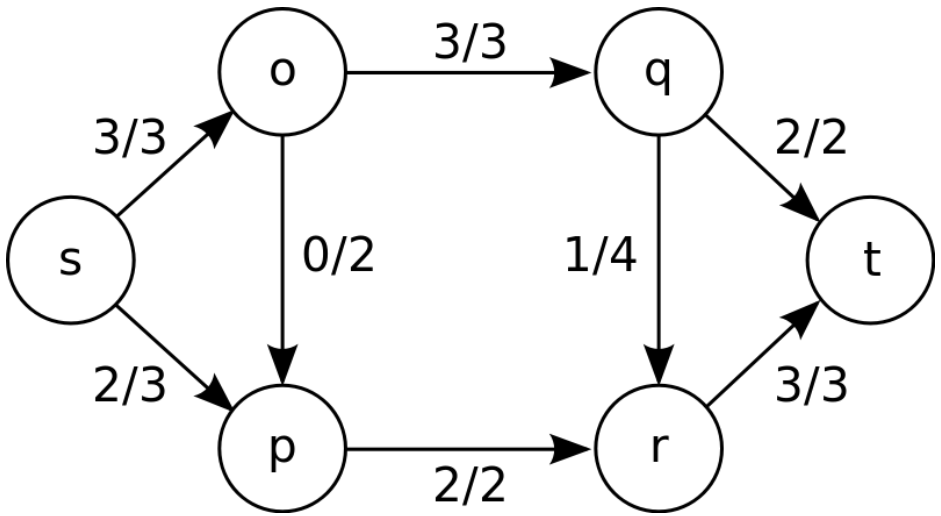
Tricks with ASPS

- 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;

Network Flow

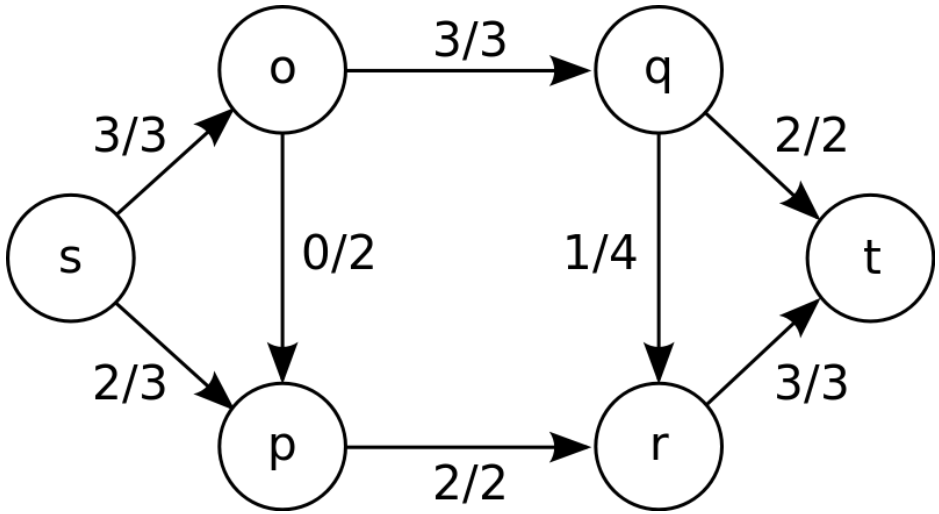
Problem Definition

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

1. Create residual graph with flow capacity between nodes;
2. Initial residual graph = directed 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. Do 3 again until you cannot find a path anymore;

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) {...}
```

You can use any path algorithm here (BFS, DFS, Djikstra, etc). Depends on your graph!

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.

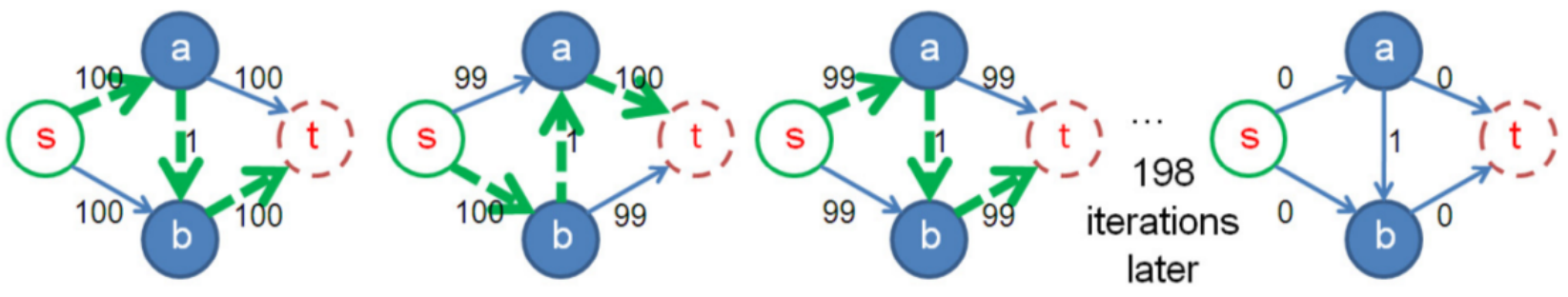


Image from Competitive Programming, 3rd edition

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;

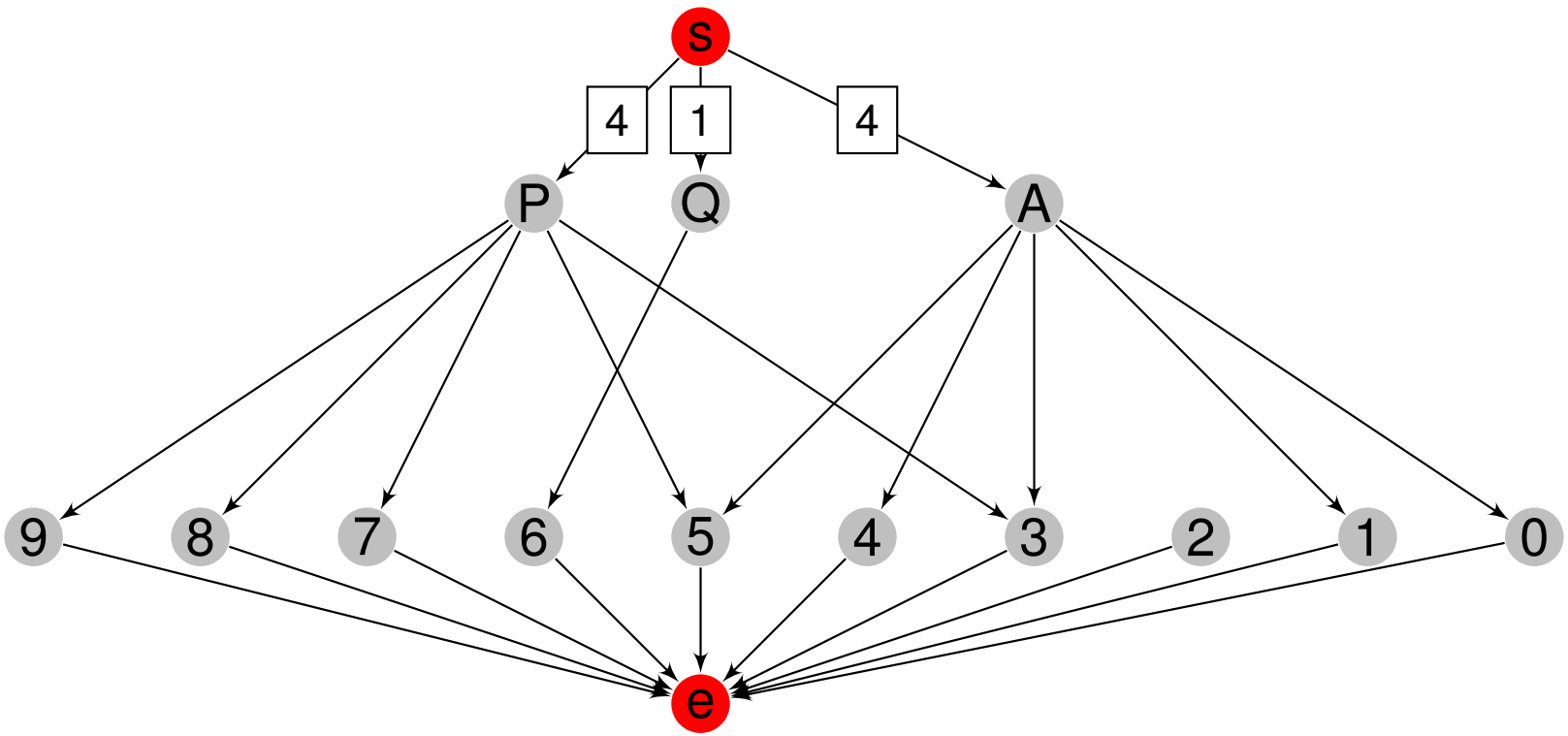
Can you find an allocation to run all programs?

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?

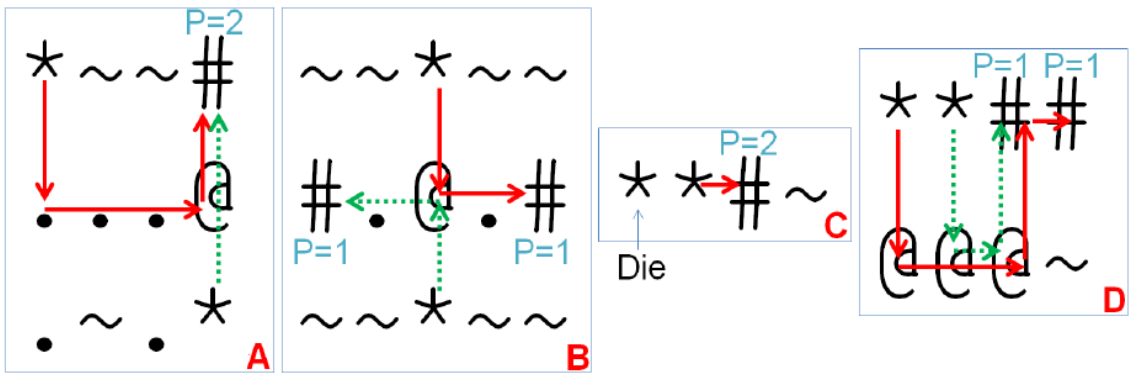


Image from *Competitive Programming, 3rd edition*

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

- Wormholes;
- Meeting Professor Miguel;
- Full Tank?;
- Degrees of Separation;
- Avoiding your Boss;
- Software Allocation;
- Sabotage;
- Gopher II;

Problem Hints

Wormholes

- **Problem goal:** Find a negative weight loop in the graph
- **Hint:** Just follow the suggestions from the class

Meeting Professor Miguel

- **Problem goal:** Find the shortest path from the student to the professor.
- **Trick:** Some edges only the student can walk, some edges only the professor can walk;
- **Hint 1:** There are really two graphs: One for the student, one for the professor;
- **Hint 2:** The graphs are really small;

Problem Hints

Full Tank?

- **Problem goal:** Find the path with least cost from s to t;
- **Trick:** The cost is the value of the fuel in the nodes, not the edges;
- **Hint 1:** Just follow the suggestions from the class
- **Hint 2:** for problems with costs on nodes, not edges, we can usually modify the graph by breaking the node into Node-Edge-Node.

Degrees of Separation

- **Problem goal:** What is the largest path inside the graph?
- **Hint 1:** Easy problem - just find all paths in the graph, and choose the largest one.
- **Hint 2:** Special case if the graph is not connected!

Problem Hints

Avoiding Your Boss

- **Problem Goal:** Find the shortest path that does not meet your boss path;
- **Hint 1:** How can you modify the graph to avoid your boss?

Software Allocation

- **Problem Goal:** Find an allocation of program instances to free computers
- **Hint:** Use the Max Flow algorithm discussed in class

Problem Hints

Sabotage

- **Problem Goal:** Find the cost of the bipartite flow
- **Hint:** Standard max flow algorithm discussed in class

Gopher II

- **Problem Goal:** Find a strategy that saves as many gophers as possible;
- **Hint:** Also bipartite flow variation
- **Hint:** Example of Implicit graph: what are the nodes? What are the edges?