

# Programming Challenges (GB21802)

## Week 5 - Graph Part II: Minimum Path

Claus Aranha

caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

2020/6/02

(last updated: May 31, 2020)

Version 2020.1

# Lecture Outline

In this lecture, we will study algorithms focused on the properties of **paths in weighted graphs**:

- Single Source Shortest Path;
  - Dijkstra Algorithm
  - Bellman-Ford Algorithm
- All Pairs Shortest Path;
  - Floyd-Warshall Algorithm
- Network Max Flow;
  - Ford-Fulkerson Method
  - Edmond-Karp Algorithm

Finally, we will also see several graph problems based on these algorithms.

# SSSP: Single Source Shortest Path

## Problem Definition

In a graph  $G(V, E)$ , find the path from vertex  $v_s$  (source) to vertex  $v_t$  (target), where **the sum of edge weights is minimal**.

- For an **unweighted graph**, we consider the weight of every edge  $e_i \in E$  to be equal to one;
- In this case, Breadth First Search (BFS) is actually good enough;
- Start BFS on node  $v_s$ ;
- BFS will visit all nodes in the graph in order of distance, eventually reaching  $v_t$  (if it is reachable);

# BFS Implementation for SSSP

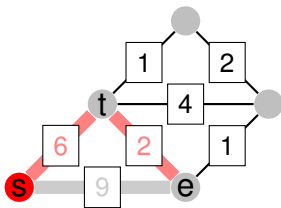
```
vector <int> p; // parent list
vector <int> dist(V,100*V); dist[s] = 0; // dist matrix
queue <int> q; q.push(s);

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] > V) { // 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

The BFS is a simple and fast algorithm. However, if the graph has edge weights, it may give you a Wrong Answer to find the SSSP



- BFS shortest path:  $s \rightarrow e$  (1 edge, cost 9)
- Real shortest path:  $s \rightarrow t \rightarrow e$  (2 edges, cost 8)

# SSSP on weighted graphs: Dijkstra's Algorithm

## Basic Idea:

Greedy graph search: Always follow the edge with the **minimal total distance** from the source node.

- There are many different implementations;
  - (The original paper did not include an implementation!)
- Simple implementation: replace the BFS **queue** with a **Priority Queue**:
  - The priority queue sorts the edges with minimum total distance;
- Minor optimization:
  - C++ STL priority queue has large cost to deleting/updating edges;
  - To avoid this cost, we use "lazy deletion" to skip longer paths;

# Dijkstra's Algorithm Implementation Example

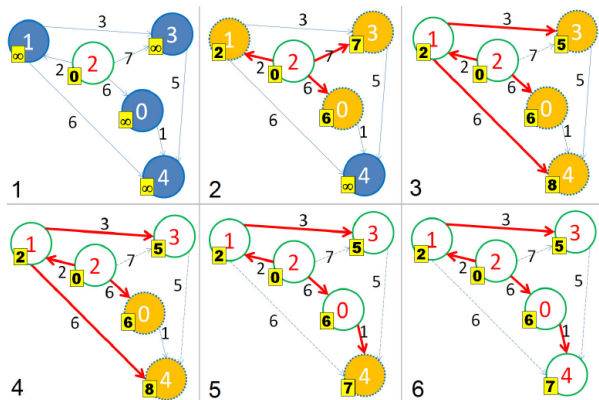
This implementation uses **Lazy Skipping** to reduce the number of deletions from the priority queue;

```
typedef pair<int,int> ii;           // <distance, to_vertex>
priority_queue<ii, vector<ii>, greater<ii>> pq;
pq.push({0,s});

while (!pq.empty()) {
    auto [d, u] = pq.top(); pq.pop(); // shortest unvisited u
    if (d > dist[u]) continue;        // **Lazy skipping**
    for (auto &[v, w] : AdjList[u]) { // all edges from u
        if (dist[u] + w >= dist[v]) continue;
        // new edge does not improve solution, skip
        dist[v] = dist[u] + w          // update distance
        pq.push({dist[v], v})         // enqueue better pair
    }
}
```

# Dijkstra with Lazy deletion: Simulation

Dijkstra visits vertices: 2, 1, 3, 0, 4; in order



PQ(0) :

PQ(1) :



# SSSP in Programming Challenges

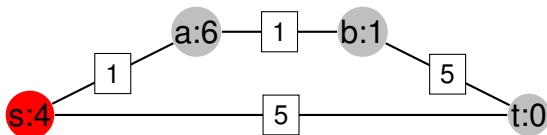
- Of course, just giving a graph as input, and asking you to find the SSSP is not a very exciting programming challenge.
- Because of this, programming challenges will usually require you to **change or build the graph structure** from the input data.
- Let's see one example;

# UVA 11367 – Full Tank

## Problem Summary

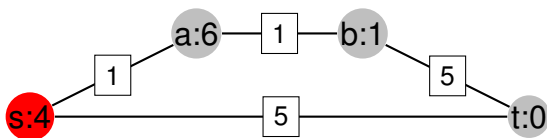
Find the **cheapest** path from city  $S$  to city  $T$ . Consider the following:

- To go from  $v_i$  to  $v_j$  requires  $E_{i,j}$  **liters of fuel**;
- The price of fuel in city  $v_i$  is  $p_i$ ;
- Your car has maximum capacity  $c$ ;



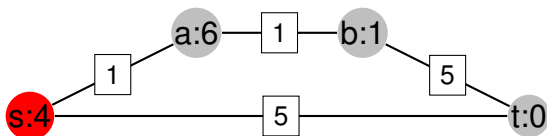
- Path:  $s \rightarrow t$ : Buy 10 liters at  $s$ , cost: 20
- Path:  $s \rightarrow a \rightarrow b \rightarrow t$ :
  - Buy 4 liters at  $s$ , 10 liters at  $b$ , cost: 9
- **QUIZ**: How to implement SSSP for this one?

# UVA 11367 – Full Tank: Graph Modification



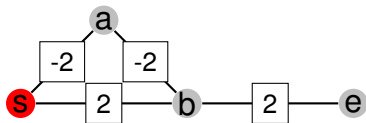
- Transform vertex  $v_i$  into a set of vertices  $v_{i,f}$ :  $v_{i,0}, v_{i,1}, \dots, v_{i,c}$ ;
  - This represents the car at  $v_i$  with  $f$  fuel left;
- An edge exists between  $v_{i,k}$  and  $v_{i,k+1}$  with cost  $p_i$ ;
  - This represents adding fuel to the car.
- An edge exists between  $v_{i,k}$  and  $v_{j,k-w_{i,j}}$  if exists an edge  $v_i \rightarrow v_j$  with cost  $w_{i,j}$  and  $k - w_{i,j} \geq 0$ ;
  - This represents the car has enough fuel to go to  $j$
- Now we can do Dijkstra in the modified graph;
- Note the new graph has  $V \times C$  vertices and  $E \times C$  edges;

# UVA 11367 – Full Tank: Simulation



# A Problem with Dijkstra

The dijkstra implementation that we discussed will fall into an **infinite loop** if the graph includes a **negative loop**!



- Our Dijkstra implementation will add smaller and smaller costs to the priority queue:
  - $s \rightarrow a$ : -2, -4, -6, -8...
- We could try to check for used edges; but that will not **detect** negative loops;
- **Bellman Ford's algorithm** is a simple SSSP algorithm that can **detect negative loops**;

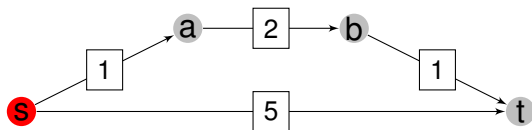
# Bellman Ford's Algorithm – ( $O(VE)$ )

- The main idea is to propagate the weight of every edge  $i \rightarrow j$ ,  $V - 1$  times.
- The vector of distances from  $s$ ,  $dist$ , starts with  $dist[s]=0$ , and  $dist[!s]=INF$ ;
- Each iteration, non-inf values of  $dist$  propagate;
- Because the algorithm has a finite number of loops, it always terminates;
- Algorithm stabilizes at iteration  $V - 1$ . If  $dist$  changes after that, we have detected an infinite loop.

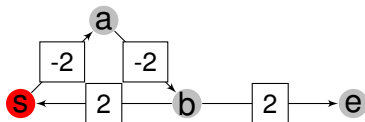
## Pseudocode (uses EdgeList data structure)

```
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]);
    }
```

# Bellman Ford Simulation: Regular Graph



# Bellman Ford Simulation: Negative Loop





# SSSP Summary

- **BFS**:  $O(V + E)$ , only for unweighted graphs;
- **Dijkstra**:  $O(E + V \log V)$ , not guaranteed to stop if there is a negative loop;
- **Bellman Ford**:  $O(EV)$  guaranteed to stop;

Be familiar with all of them, and use the most appropriate one (or simplest one) for each problem!

# APSP: All Pairs Shortest Path

Consider the following problem:

## UVA 11463 – Commandos

Consider a graph  $G(V, E)$ , with a starting vertex  $v_s$  and an end vertex  $v_t$ . You must send a group of commandos to visit every vertex in the graph.

Calculate the minimum time to complete all visits, if you can send the commandos in parallel.

**Quiz:** How do you solve this problem?

# APSP: Commandos Problem

- To solve this problem, you need to calculate, for every vertex, the shortest path from  $v_s$  to  $v_t$  that includes that vertex; The solution is the largest of these paths.
- One simple way to program this is to loop through all vertices  $v_i$ , and calculate  $\text{Dijkstra}(v_s, v_i) + \text{Dijkstra}(v_i, v_t)$ ;
  - The cost would be about  $O(V(E + V))$ ;
- Let's introduce a **simpler** (but not cheaper!) way to write this program.

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

Only four lines of code!

```
int AdjMat[V][V]; // Adjacency Matrix
// Initialization: AdjMat[i][j] contains cost
// of i->j edge, or INF if no edge.

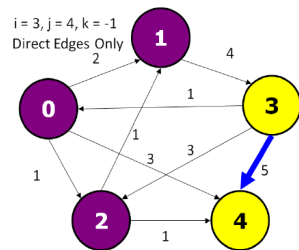
for (int k=0; k < V; k++) // loop order
    for (int i=0; i < V; i++)
        for (int j=0; j < V; j++)
            AdjMat[i][j] = min(AdjMat[i][j],
                               AdjMat[i][k]+AdjMat[k][j]);

// AdjMat[i][j]: cost of minimum path i -> j
```

- Algorithm is slower! So only use it on small graphs;
- Very easy to program: Fewer bugs!

# How does Floyd Warshall work?

- The basic idea of FW, is Bottom-up dynamic programming;
- For every vertex  $v_k$ , the shortest path between  $v_i$  and  $v_j$  is either:
  - The current shortest path  $v_i \rightarrow v_j$  or;
  - The new shortest path  $v_i \rightarrow v_k \rightarrow v_j$ ;
- Every iteration  $k$ , FW adds  $v_k$  to all other existing paths.



$sp(3, 2, -1) = 3$   $sp(2, 4, -1) = 1$   $sp(3, 4, -1) = 5$

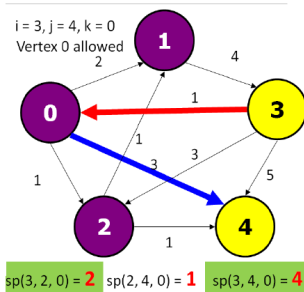
We will monitor these two values

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

$k = -1$	0	1	2	3	4
0	0	2	1	$\infty$	3
1	$\infty$	0	$\infty$	4	$\infty$
2	$\infty$	1	0	$\infty$	1
3	1	$\infty$	3	0	5
4	$\infty$	$\infty$	$\infty$	$\infty$	0

# How does Floyd Warshall work?

- The basic idea of FW, is Bottom-up dynamic programming;
- For every vertex  $v_k$ , the shortest path between  $v_i$  and  $v_j$  is either:
  - The current shortest path  $v_i \rightarrow v_j$  or;
  - The new shortest path  $v_i \rightarrow v_k \rightarrow v_j$ ;
- Every iteration  $k$ , FW adds  $v_k$  to all other existing paths.

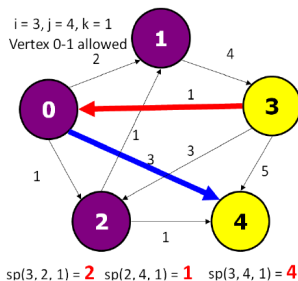


The current content of Adjacency Matrix D  
at  $k = 0$

$k = 0$	0	1	2	3	4
0	0	2	1	$\infty$	3
1	$\infty$	0	$\infty$	4	$\infty$
2	$\infty$	1	0	$\infty$	1
3	1	3	2	0	4
4	$\infty$	$\infty$	$\infty$	$\infty$	0

# How does Floyd Warshall work?

- The basic idea of FW, is Bottom-up dynamic programming;
- For every vertex  $v_k$ , the shortest path between  $v_i$  and  $v_j$  is either:
  - The current shortest path  $v_i \rightarrow v_j$  or;
  - The new shortest path  $v_i \rightarrow v_k \rightarrow v_j$ ;
- Every iteration  $k$ , FW adds  $v_k$  to all other existing paths.

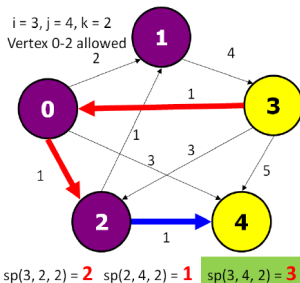


The current content of Adjacency Matrix D  
at  $k = 1$

$k = 1$	0	1	2	3	4
0	0	2	1	6	3
1	$\infty$	0	$\infty$	4	$\infty$
2	$\infty$	1	0	5	1
3	1	3	2	0	4
4	$\infty$	$\infty$	$\infty$	$\infty$	0

# How does Floyd Warshall work?

- The basic idea of FW, is Bottom-up dynamic programming;
- For every vertex  $v_k$ , the shortest path between  $v_i$  and  $v_j$  is either:
  - The current shortest path  $v_i \rightarrow v_j$  or;
  - The new shortest path  $v_i \rightarrow v_k \rightarrow v_j$ ;
- Every iteration  $k$ , FW adds  $v_k$  to all other existing paths.



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

$k = 2$	0	1	2	3	4
0	0	2	1	6	2
1	$\infty$	0	$\infty$	4	$\infty$
2	$\infty$	1	0	5	1
3	1	3	2	0	3
4	$\infty$	$\infty$	$\infty$	$\infty$	0



# Getting more from Floyd Warshall – 1

I want to print the shortest path from Floyd Warshall

To print the shortest path in FW, we add a 2D matrix  $p$ , where  $p[i][j]$  is the last node on the shortest path from  $i$  to  $j$

```
// Initialize parent matrix
for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
        p[i][j] = i;

// Floyd Warshall
for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            if (AdjMat[i][k] + AdjMat[k][j] < AdjMat[i][j]) {
                AdjMat[i][j] = AdjMat[i][k] + AdjMat[k][j];
                p[i][j] = p[k][j];          // Update parent Matrix
            }
```

# Getting more from Floyd Warshall – 2

- If we only want to know if  $v_i$  is connected to  $v_j$ , we can use FW with bitwise operations – much faster:

```
AdjMat[i][j] |= AdjMat[i][k] && AdjMat[k][j];
```

- We can use FW instead of MST to find the minmax path:

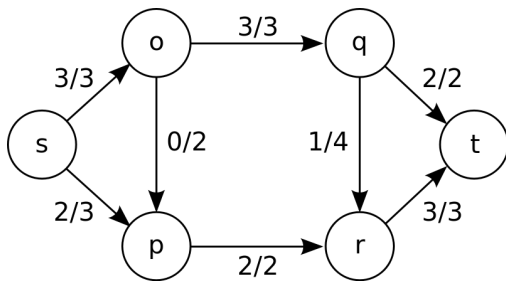
```
AdjMat[i][j] = min(AdjMat[i][j], max(AdjMat[i][k], AdjMat[k][j]));
```

- We can use FW to find SCCs:
  - If  $\text{AdjMat}[i][j] > 0$  AND  $\text{AdjMat}[j][i] > 0$ ,  $v_i$  and  $v_j$  are in same SCC;
- Use FW to detect negative cycles (or minimum cycles):
  - for  $i = 0 \rightarrow V$ , check  $\text{AdjMat}[i][i]$ ;
  - If negative: negative loop;
  - Else: minimum loop.

## Network Max Flow

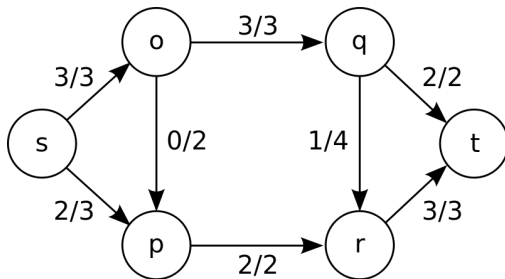
# Network Max Flow – Problem Definition

Consider a **weighted** network of pipes. The weight is the size of each pipe. Water enters the network at  $v_s$  and leave at  $v_t$ . How much water is leaving through  $v_t$ ?



- 2 units come through  $s \rightarrow o \rightarrow q \rightarrow t$ ,
- 2 units come through  $s \rightarrow p \rightarrow r \rightarrow t$ ,
- 1 unit comes through  $s \rightarrow o \rightarrow q \rightarrow r \rightarrow t$ .

# Network Max Flow – Problem Definition



The goal of the Max Flow problem is to find the maximum total flow that can go between  $v_s$  and  $v_t$  in a given graph.

# Ford Fulkerson Method for Max Flow

The Ford-Fulkerson method<sup>a</sup> finds the maximum flow using a **Residual Flow Graph** to keep track of remaining capacity.

---

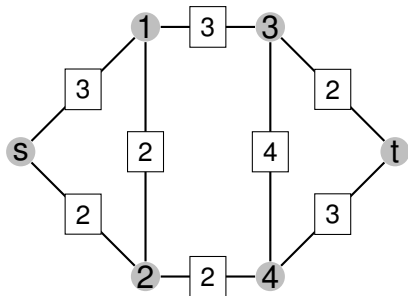
<sup>a</sup>Same Ford as in Bellman-Ford

- **Initialize Residual Graph F:** equal to the original graph  $G$ , but directed (add edges as necessary)
- **Main Loop:** If there is a path  $p$  between  $v_s$  and  $v_t$  in  $F$ :
  - Find smallest weight  $w$  in  $p$ ;
  - For every edge  $E_{u,v} \in p$ , **subtract**  $w$  from each edge;
  - For every back-edge  $E_{v,u} | E_{u,v} \in p$ , **add**  $w$  to each edge;
  - Find another path  $v_s \rightarrow v_t \in F$

# Ford Fulkerson – Pseudocode

```
// Initialize Residual Graph
int residual[V][V];
memset(residual, 0, sizeof(residual))
for (int i; i < V; i++)
    for (int j; j < V; j++)
        if residual[i][j] = AdjMat[i][j];
mf = 0; // Max flow counter
while (P = FindPath(s, t)) { // Find new path;
    m = P.min_weight; // minimum edge in P
    for (edge (v, u) in P) {
        residual[v][u] -= m;
        residual[u][v] += m;
    }
    mf += m;
}
```

# Ford-Fulkerson Simulation



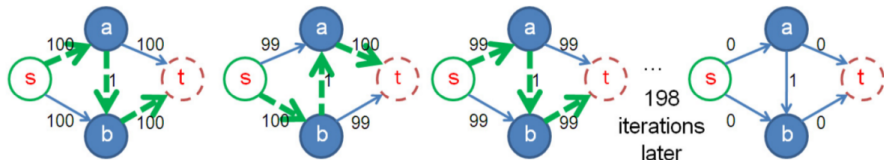


# Finding Paths in Ford Fulkerson – Problems

The Ford Fulkerson method does not specify an algorithm for finding a path in the residual graph. You could use anything!

## Problem case with bad path finding

The worst case of path selection could be  $O(|f^*|E)$ , where  $f^*$  is the true Max Flow value.



# FF efficient implementation: Edmond Karp's Algorithm

To avoid these "worst cases" of bad path selection, **Edmond Karp's** algorithm uses BFS on the residual graph to select a new  $s \rightarrow t$  path.

## Pseudocode

```
boolean BFS(s, t, p) { }
// Finds shortest (by edge #) path from s to t and store in p

mf = 0
while BFS(s,t,p) do {
  for (i in p) {
    minw = min(minw, p[i].w) // find min in p;
  }
  mf += minw;
  for (i in p) {
    res[p[i].u][p[i].v] -= minw;
    res[p[i].v][p[i].u] += minw;
  }
}
```

# UVA 259 – Software Allocation

## Outline

In a laboratory there are 26 applications and 10 computers. Each computer can run a subset of these applications. Each computer can run only one program per day.

Every day, laboratory users submit **application requests**. These requests can be repeated. For example, two users can request application A, and one user requests application B. You must determine if it is possible to satisfy all applications. If so, you must print the computer allocation.

**QUIZ:** How do you solve this program?

# UVA – Software Allocation

**Allocation Problems** (also called “matching” problems) can usually be solved using Max Flow.

The main part of the problem is: **What is the graph that best represents this problem?**

- Create a **source vertex s** connected to all applications.
  - The weight of these edges is the number of users requesting that application.
- Create an edge connecting each application to the computers that can run that application.
- Create an edge connecting each computer to a **sink vertex t**.
  - The weight of these edges is 1 (number of programs that can run on the computer).

Solve the maxflow problem for this graph. If the flow size is equal to the number of users, then the allocation is satisfied.

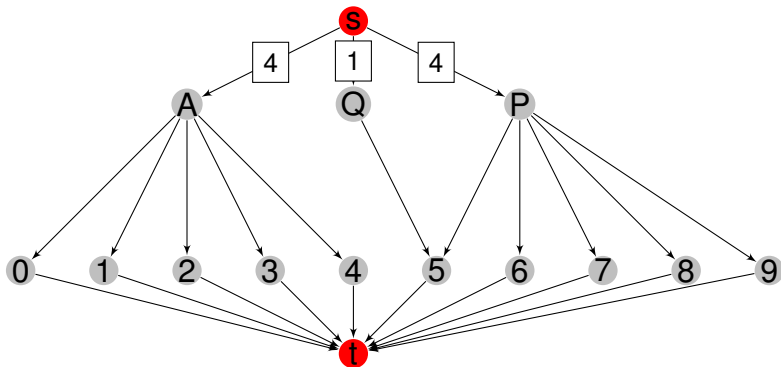
# UVA 259 – Software Allocation

## Input Example One

A4 01234;

Q1 5;

P4 56789;

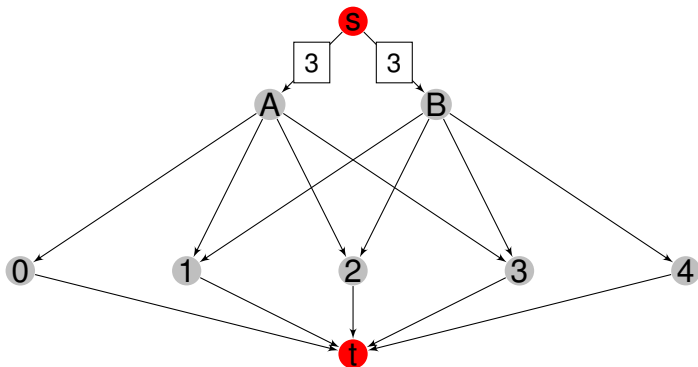


# UVA 259 – Software Allocation

## Input Example Two

A3 0123;

B3 1234;



# UVA 10480 – Sabotage

## Problem Description

Given a communication network  $V$ , what is the **minimum number of edges** that you must remove from  $V$  so that the vertices  $v_s$  and  $v_t$  are not connected?

This is a traditional graph problem called **minimum cut**. One way to solve this problem is to use the MaxFlow algorithm and analyse the **residual graph**.

- After MaxFlow, all edges in the residual graph that have weight 0 belong to the **minimum cut set**.
- A BFS on the residual graph starting from  $v_s$  will indicate the vertices that remain connected to  $v_s$  after the cut.
- The vertices not reachable in the BFS will be connected to  $v_e$ .

# Designing Network Flow Problem Graphs

## Graph with multiple sources and multiple sinks

- Create a “super source” vertex  $v_{ss}$ .  $v_{ss}$  connects to all sources with infinite weight;
- Create a “super sink” vertex  $v_{se}$ . All sinks connect to  $v_{se}$  with infinite weight;

## Graph with weights on vertices, not edges

- Similar to “full tank”, we split the graph’s vertices;
- Vertex  $v_i$  is split into  $v_{i1}$  and  $v_{i2}$ .
- Add an edge( $v_{i1}$ ,  $v_{i2}$ ) with weight  $v_i$ .
- Don’t forget that this solution doubles  $|V|$  and increases  $|E|$ .



# A few more Graph problem example

One interesting thing about graph problems is that they come in great variety.

- Many different graph algorithms;
- Many different ways to represent a problem as a graph;
- Many different problem types;

Let's see some extra problems to discuss these variations.

# Fishmonger – Shortest path on a DAG

## Problem Description

A fish seller must go from city  $v_0$  to city  $v_{n-1}$ , before the time  $t$ . The seller must also pay the minimum amount of toll.

- Every edge has a **toll cost** and a **time cost**;
- The number of cities is  $\leq 50$ , and the time  $t$  is  $\leq 1000$ .

This is a *SSSP* problem, but with two costs: *time* and *toll*.

**Quiz:** How do you find the minimal path for both costs?

# Fishmonger – Shortest path on a DAG

## Graph Transformation

Similar to "full tank", we each vertex  $v_i$  into  $t$  new vertices, where  $v_{i,t}$  indicates that you reached vertex  $v_i$  with  $t$  time left.

Each edge is modified: The edges no longer have a time cost, but they instead directionally connect  $v_{i,t} \rightarrow v_{j,t'}$ .

- This transformation multiplies the number of nodes to  $(V \times T)$ ;
- However, the new graph is now a DAG;
- Because the graph is a DAG, it is possible to solve the search using **Top-Down Dynamic Programming**:
  - Recursive function:  $find(v,t)$ , finds minimum cost  $c$
  - Table size:  $V \times T$
- In fact, it is not even necessary to *explicitly* transform the graph. An implicit transformation inside the recursive function is enough!

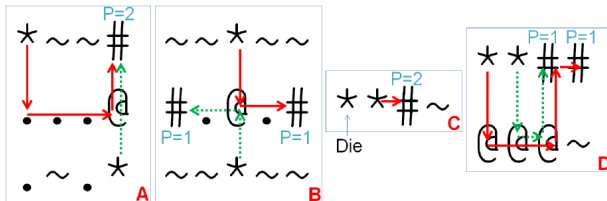
# Titanic (UVA 11380) – Network Flow

## Problem Outline

Given the map of an accident, calculate how many people escape.

Map explanation:

- # -- large wood: safe, P people can escape;
- @ -- large iceberg: anyone can pass;
- . -- ice: breaks after 1 person pass;
- \* -- initial position: 1 person. Breaks like ice;
- ~ -- freezing water: no one can pass;



# Titanic (UVA11380) – Network Flow

## Graph Modeling

Number of escaped people = Max Flow of the graph. But how do we create the flow graph?

One example:

- Every non-water cell has two vertices: "In" and "Out";
- Connect "In" and "Out" vertices with cell capacity:
  - Iceberg and wood: Infinite
  - ice and initial position: 1
- Connect every "Out" vertex with other nearby "In" vertices with infinite capacity;
- Connect "Super Source" with all "initial positions" with capacity 1;
- Connect all "Wood" with "Super Sink", with capacity P.

# Prime Pairing – Bipartite Graph Flow

## Problem Description

Two numbers  $a, b$  are be **prime paired** if  $a + b$  is prime.

Given a set of numbers  $N$ , is it possible to create a **complete pairing** with all elements of  $N$ ?

Example:

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

Is this even a graph problem??

# Prime Pairing – Bipartite Graph Flow

## Trick

It is possible to think of this problem as an **allocation** problem.

Remember that **even + even = even** and **odd + odd = even**. So a prime pair must be one even # and one odd #.

In this way, we must allocate even numbers to odd numbers (or vice-versa)

How to create the graph:

- Split set between odds and evens;
- If  $\#odd$  is not equal to  $\#even$ , there is no solution;
- Create edges between odds and evens if they are a prime pair;
- Add a super source and super sink;

If  $\text{max flow} = \# \text{ vertices} / 2$ , then there is a solution.

# Lecture Summary

## Graph Algorithms for Path Finding and Maximum Flow:

- Single Source Shortest Path:
  - In an unweighted graph, use BFS;
  - For a weighted graph, use Dijkstra;
  - If the graph has negative loops, Bellman-ford will terminate;
- All Pairs Shortest path:
  - Floyd-Warshall is very easy to program, but costs  $O(V^3)$ ;
  - You could also just repeat Dijkstra  $V$  times;
- Maximum Flow:
  - The Ford-Fergusson Method describes how to find the maximum Flow;
  - Edmond-Karp implements FF using BFS on the residual graph to find minimum paths;

The most important skill to learn for graph problems is **how to transform the problem graph**.



# 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?

- Discussed in class;
- Good practice on modifying graphs to solve problems;

## Degrees of Separation

- **Problem Outline:** Given a network of relationship, define the "Degree of Separation" of the network. "Degree of separation" is the **largest shortest path** in the network.
- **Hint 1:** Don't forget the special case of a disconnected Graph;
- **Hint 2:** The "largest shortest path" of a graph is also known as the **diameter**, and it is an important property of graphs;

# Problem Hints

## Avoiding Your Boss

- **Problem Goal:** You are going from your house to the market. Your boss is going from her house to the office. Can you find a path that does not meet your boss' path?
- **Hint:** How do you modify the graph so that you can avoid your boss?

## Software Allocation

- Discussed in Class;
- Use this problem to practice "Max Flow";

# Problem Hints

## Sabotage

- **Problem Goal:** Find the cost of minimum cut;
- Discussed in class, use Max Flow to find the minimum cut set;

## Gopher II

### Problem Outline:

- You receive the position of  $N$  gophers and  $M$  gopher holes.
- Each hole can save 1 Gopher, Gophers run at speed  $v$ .
- Hawks will eat the Gophers after  $t$  seconds;
- How many gophers are saved? How many are eaten?

### Hints:

- Think of an **Allocation** of gophers to holes;
- Graph is implicit: How do you define the edges?

# About these Slides

These slides were made by Claus Aranha, 2020. You are welcome to copy, re-use and modify this material.

Individual images in some slides might have been made by other authors. Please see the references in each slide for those cases.

# Image Credits I

[Page 8] Dijkstra Image from "Competitive Programming 3", Steven Halim

[Page 21] Floyd-Warshall Image from "Competitive Programming", Steven Halim

[Page 21] Floyd-Warshall Image from "Competitive Programming", Steven Halim

[Page 21] Floyd-Warshall Image from "Competitive Programming", Steven Halim

[Page 21] Floyd-Warshall Image from "Competitive Programming", Steven Halim

[Page 25] Network Flow Image CC-BY-SA 3.0 by Maksim

[Page 30] Image from "Competitive Programming 3", Steven Halim

[Page 41] Image from *Competitive Programming, 3rd edition*