GB20602 - Programming Challenges

Week 6 - Graph Part II: Minimum Path

Claus Aranha

caranha@cs.tsukuba.ac.jp

University of Tsukuba, Department of Computer Sciences

(last updated: May 30, 2022)

Version 2021.1

Lecture Outline

In this material covers the following graph algorithms:

- **Part I**: Djikstra (Shortest Path);
- **Part II**: Bellman-Ford (Negative Loops);
- **Part III**: Floyd-Warshall (All Pairs Shortest Path)
- **Part IV**: Ford-Fulkerson/Edmond-Karp (Max Flow)

Part I - Djikstra (Shortest Path)

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), with **minimum sum of weights.**

- If the graph is **unweighted** (every weight is equal), use Breadth First Search (BFS).
- Start BFS on node v_s;
- BFS visits all vertices by order of distance, and reach v_t with minimum steps;

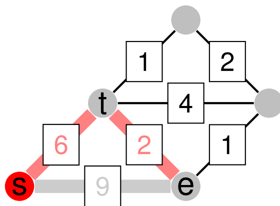
BFS Implementation for SSSP

```
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 does not work with weighted graphs

BFS is simple and fast when the graph is **unweighted**.

But when the graph has weights, BFS gives you **wrong answer**.



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

SSSP on weighted graphs: Dijkstra's Algorithm

Basic Idea: Greedy Graph Search

Always visit the vertex with **minimal total distance** from the source vertice.

- There are many different implementations;
  - (The original paper did not include a specific implementation!)
- Simple implementation: replace the BFS **queue** with a **Priority Queue**:
  - The priority queue sorts the edges by total distance from source;
- Note: Lazy Deletion Optimization
  - C++ STL priority queue has large cost to deleting/updating edges;
  - To avoid this cost, we do not delete edges (but skip them if necessary);

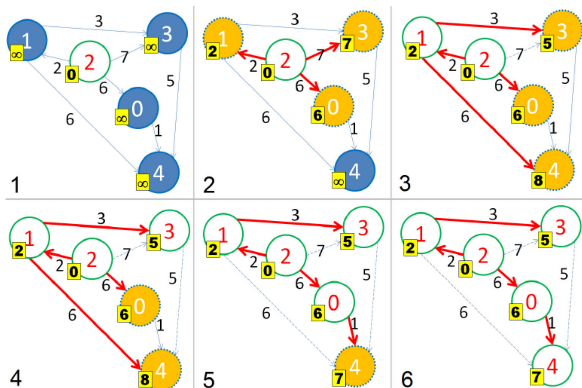
Dijkstra's Algorithm Implementation Example

This implementation uses **Lazy Deletion** to reduce the number of deletions in 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;         // skip edges that don't improve the path
  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:

SSSP and Programming Challenges

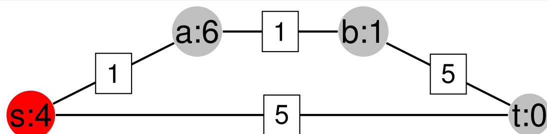
- Use the **Practice Problem** to train the implementation of Djikstra;
- In Programming Challenges, a big part of the problem is to build the correct graph;
- Think about the correct **vertices, edges and weights** from the input data;

Example Problem – Full Tank

Problem Summary

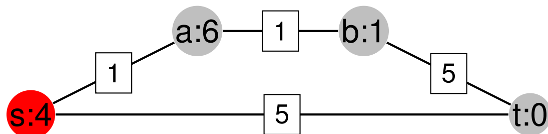
Find the **cheapest** path from city S to city T. Consider the following:

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



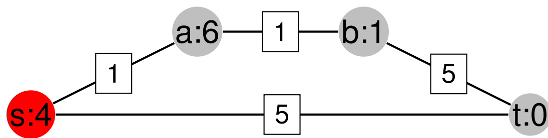
- Path: $s \rightarrow t$: Buy 10 liters at s, cost: 20
- Path: $s \rightarrow a \rightarrow b \rightarrow t$:
  - Buy 4 liters at s, 10 liters at b, cost: 9
- **QUIZ**: What is the correct graph structure to solve this problem?

Example Problem – Full Tank: Building the Graph



- Transform vertex v_i into a set of vertices $v_{i,f}$: $v_{i,0}, v_{i,1}, \ldots, v_{i,c}$;
  - 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 $v_i \rightarrow v_j$ in the original graph with cost $w_{i,j}$
  - $k - w_{i,j} \geq 0$;
  - This represents the car has enough fuel to go from i 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;

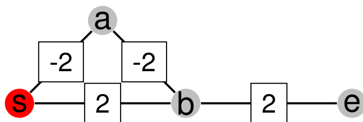
Example Problem – Full Tank: Simulation of Graph Transformation



Part II - Bellman-Ford (Negative Loops);

A Problem with Dijkstra

Djikstra can have difficulty when 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...
- Other implementations will have different problems because negative loop breaks the **Greedy Property.**
  - But what is the size of the path when a negative loop exists?
- **Bellman Ford's algorithm** is a slower SSSP algorithm that **detects negative loops**;

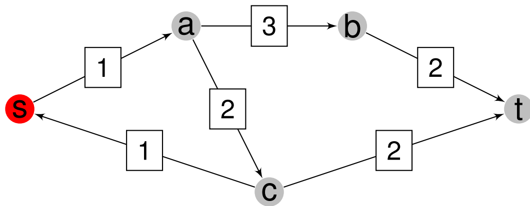
Bellman Ford's Algorithm – ($O(VE)$)

- The main idea is to propagate the weight of every edge $i \to j$, $V - 1$ 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.

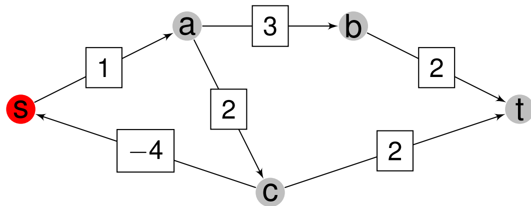
Pseudocode (uses EdgeList data structure)

```
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



Part III: Floyd-Warshall (All Pairs Shortest Path)

APSP: All Pairs Shortest Path

Consider the following problem:

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 commands 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 each vertex v_i, the shortest path $v_s \rightarrow v_i \rightarrow v_t$. The solution is the largest of these paths.
- One simple way to program this is to loop through all vertices v_i, and calculate Dijkstra(v_s, v_i) + Dijkstra(v_i, v_t);
  - The cost would be about $O(V(E + V))$;
- There is a **simpler** (but not cheaper!) algorithm to solve this problem.

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

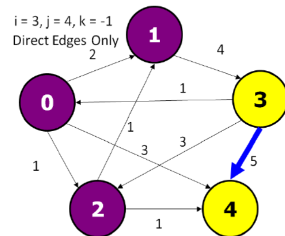
Only four lines of code!

```
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**

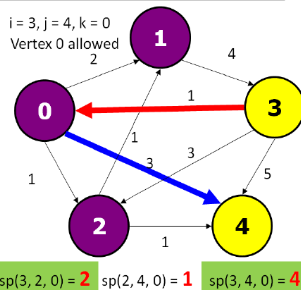
We will monitor these two values

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

k = -1	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	**5**
4	∞	∞	∞	∞	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 $v_i \to v_j$ or;
  - The new shortest path $v_i \to v_k \to v_j$;
- Every iteration k, FW adds v_k to all other existing paths.

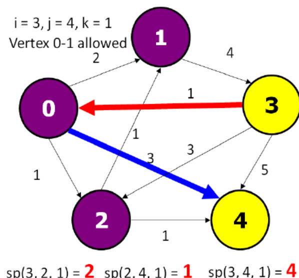


The current content of Adjacency Matrix D
at **k = 0**

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

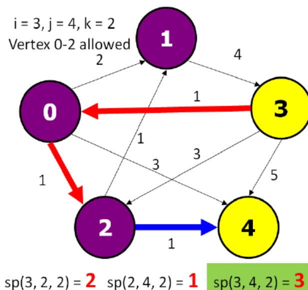


The current content of Adjacency Matrix D
at **k = 1**

k = 1	0	1	2	3	4
0	0	2	1	6	3
1	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	**4**
4	∞	∞	∞	∞	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 $v_i \to v_j$ or;
  - The new shortest path $v_i \to v_k \to v_j$;
- 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	∞	0	∞	4	∞
2	∞	1	0	5	1
3	1	3	2	0	3
4	∞	∞	∞	∞	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:

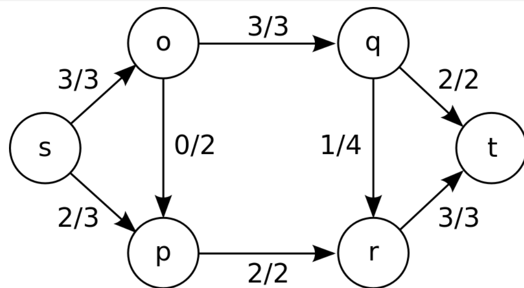
```
AdjMat[i][j]=min(AdjMat[i][j],max(AdjMat[i][k],AdjMat[k][j]));
```

- We can use FW to find SCCs:
  - If AdjMat[i][j] $>$ 0 AND 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 AdjMat[i][i];
  - If negative: negative loop;
  - Else: minimum loop.

Part IV - Network Max Flow

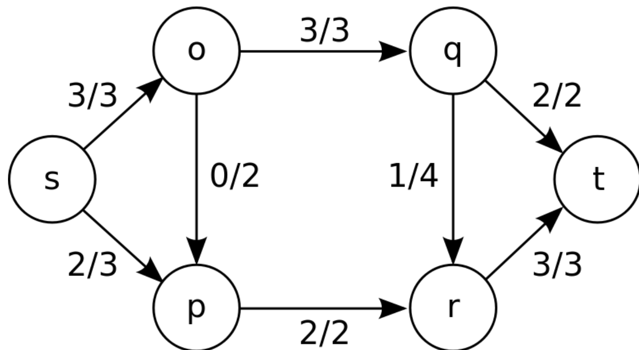
Network Max Flow – Problem Definition

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

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[a] finds the maximum flow using a **Residual Flow Graph** to keep track of remaining capacity.

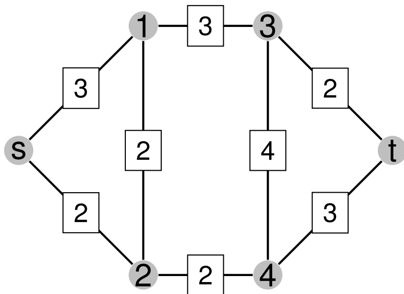
[a] Same Ford as in Bellman-Ford

- **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

```
int residual[V][V];                          // Initialize Residual Graph
memset(residual, 0, sizeof(residual))
for (int i; i < V; i++)
  for (int j; j < V; j++)
    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

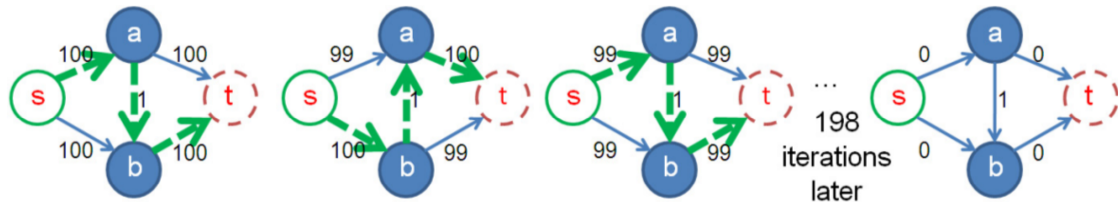


Finding Paths in Ford Fulkerson – Problems

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 $s \to t$ path.

Pseudocode

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

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;
  }
}
```

Example: 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?

Example: 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?**

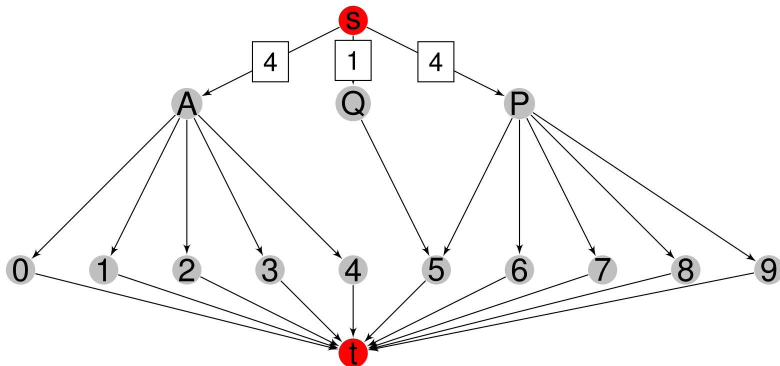
- Create a **source vertex s** connected to all applications.
  - The weight of these edges is the number of users requesting that application.
- Create an edge connecting each application to the computers that can run that application.
- Create an edge connecting each computer to a **sink vertex t**.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

Example: Software Allocation

Input Example One

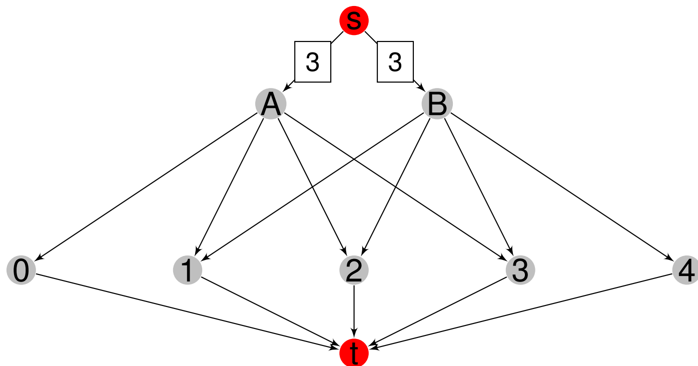
A4 01234;
Q1 5;
P4 56789;



Example: Software Allocation

Input Example Two

A3 0123;
B3 1234;



Example 2: 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 $|E|$.

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 max flow = # vertices / 2, then there is a solution.

About these Slides

These slides were made by Claus Aranha, 2022. You are welcome to copy, distribute, re-use and modify this material. (CC-BY-4.0)

Individual images in some slides might have been made by other authors. Please see the following pages for details.

Image Credits I

[Page 9] Dijkstra Image from "Competitive Programming 3", Steven Halim
[Page 23] Floyd-Warshall Image from "Competitive Programming", Steven Halim
[Page 23] Floyd-Warshall Image from "Competitive Programming", Steven Halim
[Page 23] Floyd-Warshall Image from "Competitive Programming", Steven Halim
[Page 23] Floyd-Warshall Image from "Competitive Programming", Steven Halim
[Page 27] Network Flow Image CC-BY-SA 3.0 by Maksim
[Page 32] Image from "Competitive Programming 3", Steven Halim