This course material is now made available for public usage.
Special acknowledgement to School of Computing, National University of Singapore
for allowing Steven to prepare and distribute these teaching materials.

CS3233
Competitive Programming

Dr. Steven Halim

Graph “Basics”

icpc international collegiate programming contest
icpc.foundation

Singapore 2020

32ND INTERNATIONAL OLYMPIAD IN INFORMATICS
SINGAPORE

Outline (1)

- Graph: Preliminary & Motivation
- Graph Data Structures
  - Adjacency Matrix, Adjacency List, Edge List, and Implicit Graph
- Graph Traversal Algorithms
  - DFS/BFS: Connected Components/Flood Fill
  - DFS only: Cycle Check/Toposort~~/Cut Vertex+Bridges/Strongly CCs~~
- Minimum Spanning Tree Algorithms
  - Kruskal's ~~and Prim's~~

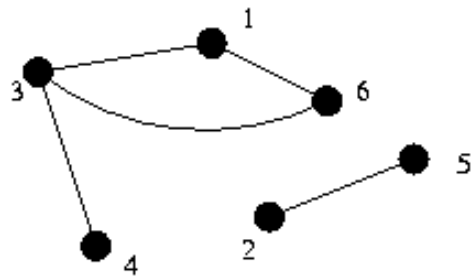
Outline (2)

- Single-Source Shortest Paths
  - BFS: SSSP on Unweighted graph/Variants
  - Dijkstra's: SSSP on Weighted (no -ve cycle) graph/Variants
  - ~~Bellman Ford's~~
- All-Pairs Shortest Paths
  - Floyd Warshall's
- Special Graphs
  - Tree, Directed Acyclic Graph~~, Bipartite Graph, Euler Graph~~

Graph Terminologies

- Vertices/Nodes
- Edges/Links
- Un/Weighted
- Un/Directed
- In/Out Degree
- Self-Loop+Multiple Edges (Multigraph)/Simple Graph
- Sparse/Dense
- Path, Cycle
- Isolated, Reachable

- (Strongly) Connected Component
- Sub Graph
- Complete Graph
- Directed Acyclic Graph
- Tree/Forest
- Euler/Hamiltonian Path/Cycle
- Bipartite Graph



CP4 Section 2.4.1

GRAPH DATA STRUCTURES

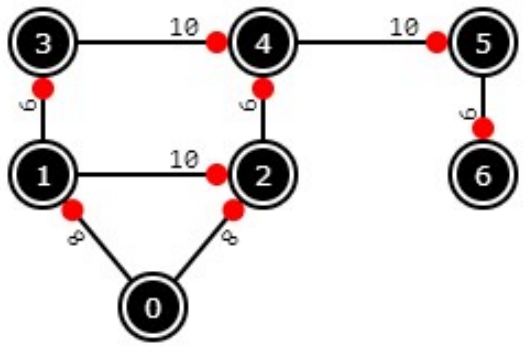
https://visualgo.net/en/graphds

Graph Data Structures (1)

Graph is a special data structure used to model objects and their connections…

Graph Representations:

- int AM[V][V]; // AM = AdjacencyMatrix, AL = Adjacency List
- vector<vii> AL; // ii is pair<int, int>, vii is vector<ii>
- vector< pair<int, ii> > EL; // EL = Edge List



Adjancency Matrix

	0	1	2	3	4	5	6
0	0	8	8	0	0	0	0
1	0	0	10	6	0	0	0
2	0	0	0	0	6	0	0
3	0	0	0	0	10	0	0
4	0	0	0	0	0	10	0
5	0	0	0	0	0	0	6
6	0	0	0	0	0	0	0

Adjacency List

```
0: (1, 8) (2, 8)
1: (2, 10) (3, 6)
2: (4, 6)
3: (4, 10)
4: (5, 10)
5: (6, 6)
6:
```

Edge List

```
0: 8  0 1
1: 8  0 2
2: 10 1 2
3: 6  1 3
4: 6  2 4
5: 10 3 4
6: 10 4 5
7: 6  5 6
```

Graph Data Structures (2)

Typical Input:

3 // n
0 2 3 // cell[i][j] > 0 implies that
0 0 4 // ∃ an edge between vertex
0 1 0 // i-j with weight cell[i][j]

Typical Input:

3 // n
2 1 2 2 3 // vertex 0 → 2 neighbors
1 2 4 // pair (neighbor, weight)
1 3 1

Adjacency Matrix:

```
const int MV = 1000 // set size
int AM[MV][MV]; // global var

// in int main
int V; scanf("%d", &V);
for (int u = 0; u < V; ++u)
  for (int v = 0; v < V; ++v)
    scanf("%d", &AM[u][v]);
```

Adjacency List (use STL):

```
vector<vii> AL;
int V; scanf("%d", &V);
AL.assign(V, vii());
for (int u = 0; u < V; ++u) {
  int k; scanf("%d", &k);
  while (k--) {
    int v, w;
    scanf("%d %d", &v, &w);
    AL[u].emplace_back(v, w);
  }
}
```

Graph Data Structures (3)

Adjacency Matrix

Pros:

- Existence of edge i-j can be found in O(1)
- Good for dense graph/ Floyd Warshall's*

Cons:

- O(V) to enumerate neighbors of a vertex
- $O(V^2)$ space

Adjacency List

Pros:

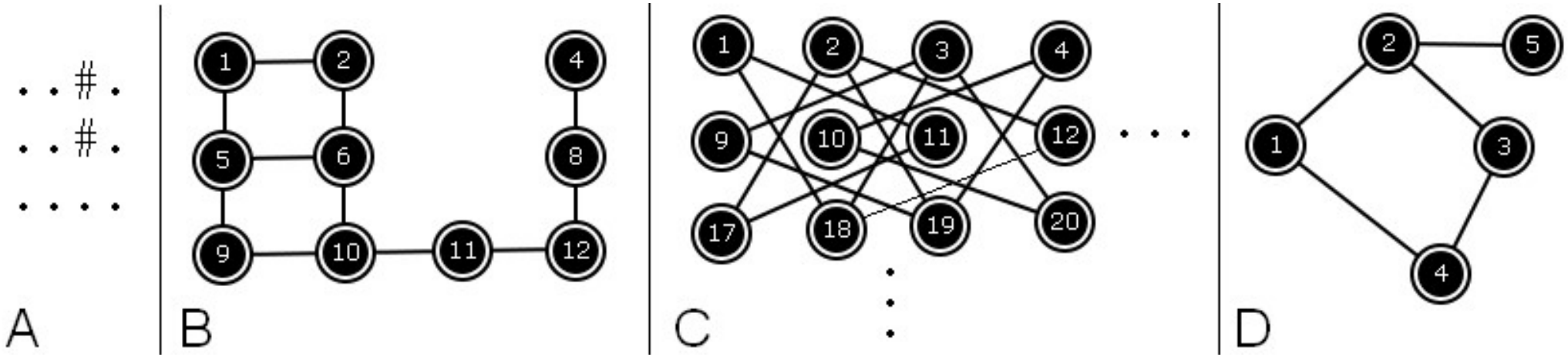
- O(k) to enumerate k neighbors of a vertex
- Good for sparse graph/ Dijkstra's*/DFS/BFS

Cons:

- O(k) to check the existence of edge i-j

Implicit Graph

- Found in some (harder) problems
- The edges can be determined easily
  - e.g. 2D grid, Knight movement in a chessboard
- The edges can be determined with some rules
  - e.g. Connect two vertices i and j if (i + j) is a prime



CP4 Section 4.2

Depth-First Search (DFS)

- Finding Connected Components
- Flood Fill
- Cycle Check (Detect Back Edge)
- Topological Sort
- ~~Bipartite Graph Check~~
- ~~Finding Articulation Points & Bridges~~
- ~~Finding Strongly Connected Components~~

Breadth-First Search (BFS)

GRAPH TRAVERSAL ALGORITHMS

Motivation (1)

How to solve these problems:

- UVa 469 (Wetlands of Florida)
  - Similar problems: UVa 260, 352, 572, 782, 784, 785, etc
- Kattis – reachableroads
  - Similar problems: gold, coast, brexit, etc
- IOI 2011 – Tropical Garden

Without familiarity with **Depth-First Search** algorithm and its variants, they look "hard"

Motivation (2)

How to solve these problems:

- UVa 336 (A Node Too Far)
  - Similar problems: UVa 383, 439, 532, 762, 10009, etc
- Kattis grid
  - Similar problems: oceancurrents, lava, etc
- IOI 2009 – Mecho

Without familiarity with **Breadth-First Search** graph traversal algorithm**,** they look "hard"

Graph Traversal Algorithms

Given a graph (in a DS), we want to traverse it!

There are 2 major ways:

1. Depth First Search (DFS)
   - Naturally implemented using recursion
   - Similar (but not the same) to "recursive backtracking"
   - Most frequently used to traverse a graph
2. Breadth First Search (BFS)
   - Usually implemented using queue, use STL
   - Can solve special case* of "shortest paths" problem!

     * unweighted

Depth First Search – Template

O(V + E) if using Adjacency List

But $O(V^2)$ if using Adjacency Matrix

```
typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;

void dfs(int u) {                                  // normal usage
  printf(" %d", u);                                // this vertex is visited
  dfs_num[u] = VISITED;                            // mark u as visited
  for (auto &[v, w] : AL[u])                       // C++17 style, w ignored
    if (dfs_num[v] == UNVISITED)                   // to avoid cycle
      dfs(v);                                      // recursively visits v
}
```

DFS != Backtracking

PS: If we undo the "visited check" part when done, DFS becomes backtracking (explores all branches)

- Slow but useful on certain cases!
  - Efficient pruning must be done!

```
void backtrack(int u) {
  dfs_num[u] = VISITED;                    // mark u as visited
  for (auto &[v, w] : AL[u])               // C++17 style, w ignored
    if (dfs_num[v] == UNVISITED)           // to avoid cycle
      backtrack(v);                        // recursively visits v
  dfs_num[u] = UNVISITED;                  // NO LONGER DFS
}
```

Breadth First Search (using STL)

O(V + E) if using Adjacency List

But $O(V^2)$ if using Adjacency Matrix

```
vi dist(V, INF); dist[s] = 0;                    // INF = 1e9 here
queue<int> q; q.push(s);

while (!q.empty()) {
  int u = q.front(); q.pop();
  printf("visit %d, ", u);
  for (auto &[v, w] : AL[u]) {                   // C++17 style, w ignored
    if (dist[v] != INF) continue;                // already visited, skip
    dist[v] = dist[u]+1;                         // dist[v] != INF now
    q.push(v);                                   // for next iteration
  }
}
```

https://github.com/stevenhalim/cpbook-code/blob/master/ch4/sssp/bfs.cpp

Connected Components

DFS (and BFS) can find connected components

- A call of dfs(**u**)/bfs(**u**) visits only the vertices connected to **u**

```
dfs_num.assign(V, UNVISITED);
int numCC = 0;
for (int u = 0; u < V; ++u)                    // for each u in [0..V-1]
  if (dfs_num[u] == UNVISITED)                 // if that u is unvisited
    printf("CC %d:", ++numCC), dfs(u), printf("\n"); // 3 lines
printf("There are %d connected components\n", numCC);
```

https://github.com/stevenhalim/cpbook-code/blob/master/ch4/traversal/dfs_cc.cpp

Variant: Flood Fill

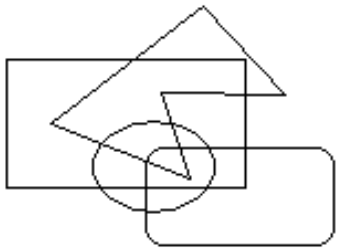
https://github.com/stevenhalim/cpbook-code/blob/master/ch4/traversal/UVa00469.cpp

Usually done on implicit graph (2D grid) - UVa 469

- Vertices: cells in grid, Edges: N/E/S/W (or to 8 directions)

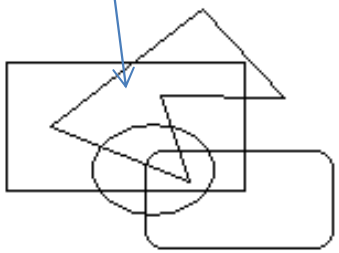
```
int dr[] = { 1, 1, 0,-1,-1,-1, 0, 1};             // the order is:
int dc[] = { 0, 1, 1, 1, 0,-1,-1,-1};             // S/SE/E/NE/N/NW/W/SW

int floodfill(int r, int c, char c1, char c2) {  // returns the size of CC
  if ((r < 0) || (r >= R) || (c < 0) || (c >= C)) return 0; // outside grid
  if (grid[r][c] != c1) return 0;                 // does not have color c1
  int ans = 1;                                    // (r, c) has color c1
  grid[r][c] = c2;                                // to avoid cycling
  for (int d = 0; d < 8; ++d)
    ans += floodfill(r+dr[d], c+dc[d], c1, c2);  // the code is neat as
  return ans;                                     // we use dr[] and dc[]
}
```

Visualization – UVa 469



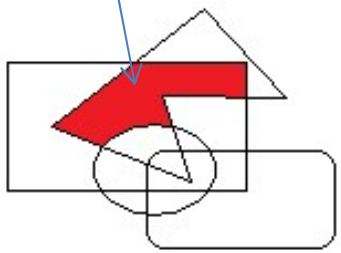
```
LLLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

Starting point



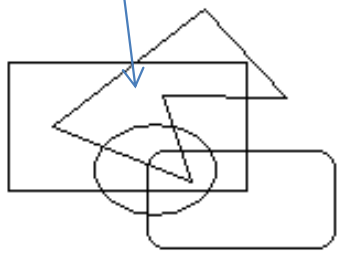
```
LLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

Flooded + area counted



```
LLLLLLLLL
LL..LLWLL
L..LLLLLL
L...L..LL
LLL...LLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

To undo the flood fill, simply reverse the color

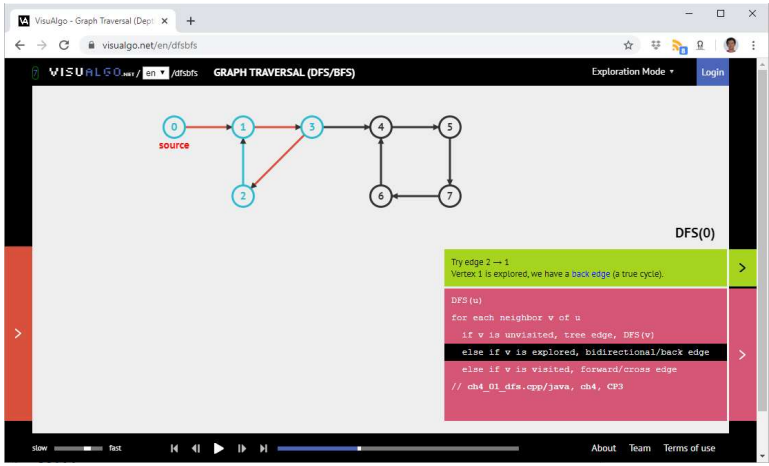


```
LLLLLLLLL
LLWWLLWLL
LWWLLLLLL
LWWWLWWLL
LLLWWWLLL
LLLLLLLLL
LLLWWLLWL
LLWLWLLLL
LLLLLLLLL
```

DFS Spanning Tree/Forest

- If DFS runs on a *connected component* of a graph, it will form a **DFS spanning tree**
  - With it, we can classify edges into four types:
    - Tree edges: those selected/traversed by DFS
    - **Back edges: for testing cycle (the most important one)**
    - Forward edges: connect vertex to its descendant
    - Cross edges: all other edges
- If the graph has many components, we have **DFS spanning forest** (for finding components)!

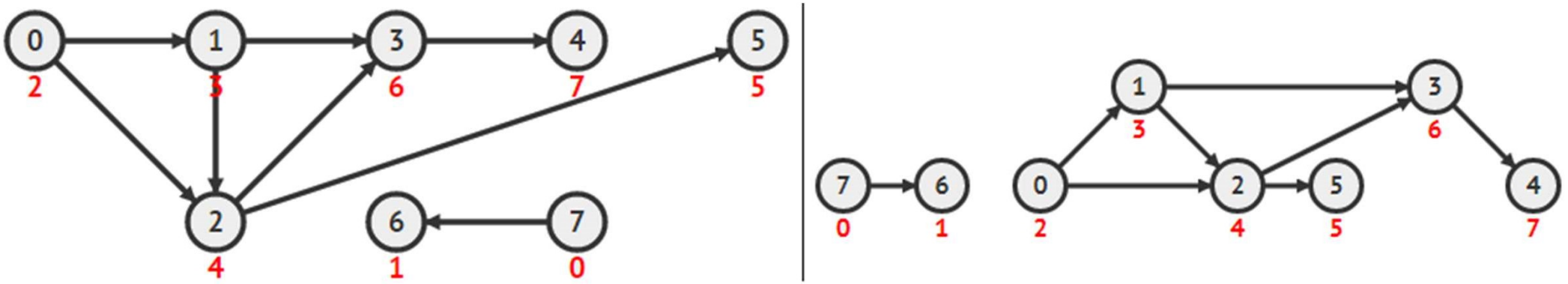
DFS, full version



https://github.com/stevenhalim/cpbook-code/blob/master/ch4/traversal/cyclecheck.cpp

```
vector<vii> AL;
vi dfs_num;
vi dfs_parent;                                  // back vs bidirectional

void cycleCheck(int u) {                        // check edge properties
  dfs_num[u] = EXPLORED;                        // color u as EXPLORED
  for (auto &[v, w] : AL[u]) {                  // C++17 style, w ignored
    if (dfs_num[v] == UNVISITED) {              // EXPLORED->UNVISITED
      dfs_parent[v] = u;                        // a tree edge u->v
      cycleCheck(v);
    }
    else if (dfs_num[v] == EXPLORED) {          // EXPLORED->EXPLORED
      if (v == dfs_parent[u])                   // differentiate them
        printf(" Bidirectional Edge (%d, %d)-(%d, %d)\n", u, v, v, u);
      else // the most frequent application: check if the graph is cyclic
        printf("Back Edge (%d, %d) (Cycle)\n", u, v);
    }
    else if (dfs_num[v] == VISITED)             // EXPLORED->VISITED
      printf("  Forward/Cross Edge (%d, %d)\n", u, v);
  }
  dfs_num[u] = VISITED;                         // color u as VISITED/DONE
}
```

Topological Sort (1)



Valid toposort: 7, 6, 0, 1, 2, 5, 3, 4
Another toposort: 0, 1, 2, 5, 3, 4, 7, 6

Q: Can you find another valid toposort?

Topological Sort (2)

Just a simple modification from standard DFS

- Append currently visited node to list of visited nodes only after processing all its children
  - This satisfies the topological sort property

```
void toposort(int u) {
  dfs_num[u] = VISITED;
  for (auto &[v, w] : AL[u])
    if (dfs_num[v] == UNVISITED)
      toposort(v);
  ts.push_back(u);                          // this is the only change
}
```

Graph Traversal Comparison

DFS

Pros:

- 'Shorter' to code
- Use less memory

Cons:

- Cannot solve SSSP on unweighted graphs

BFS

Pros:

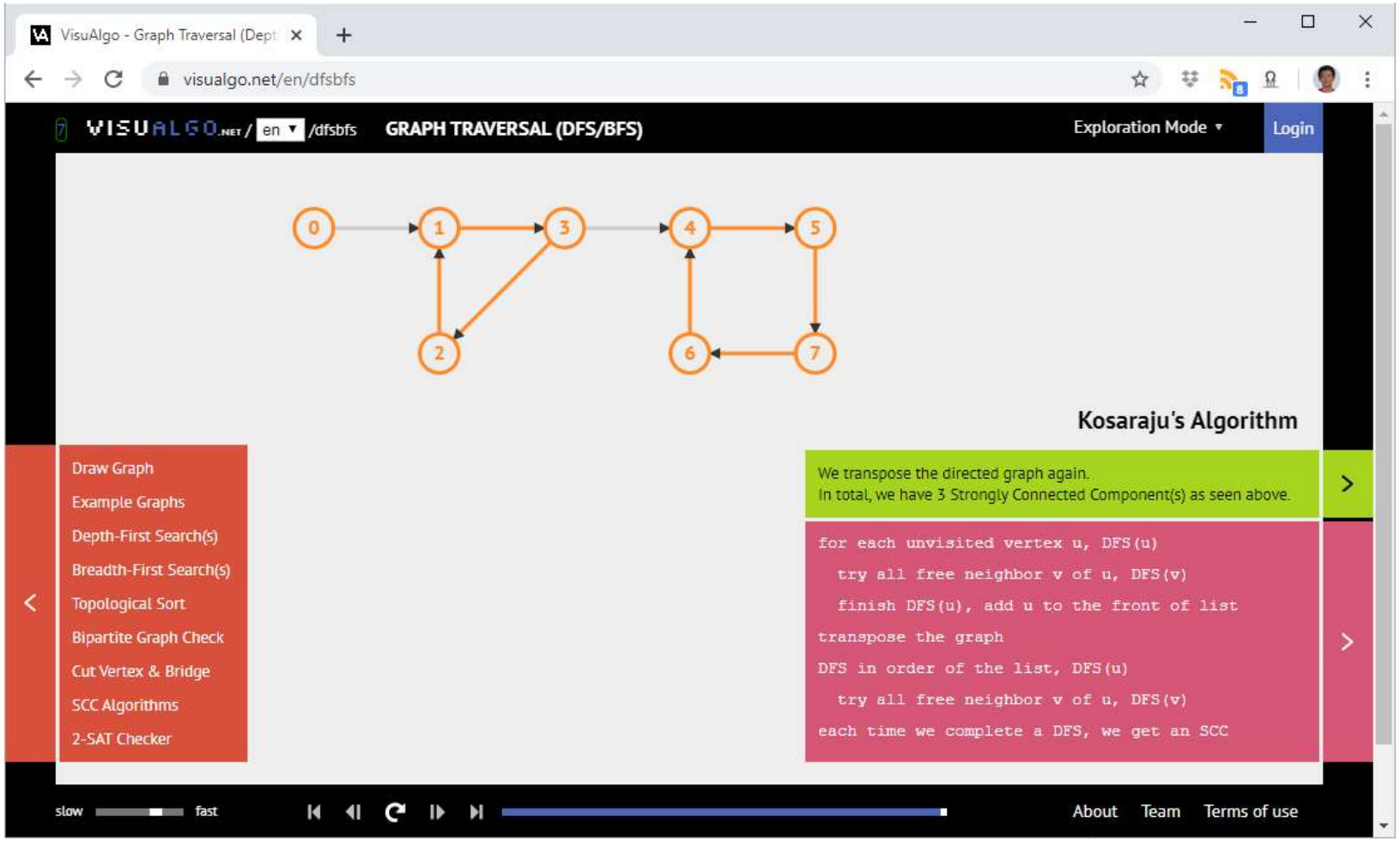
- Can solve SSSP on unweighted graphs (discussed later)

Cons:

- 'Longer' to code
- Use more memory

DFS/BFS Animation (see website)

https://visualgo.net/en/dfsbfs

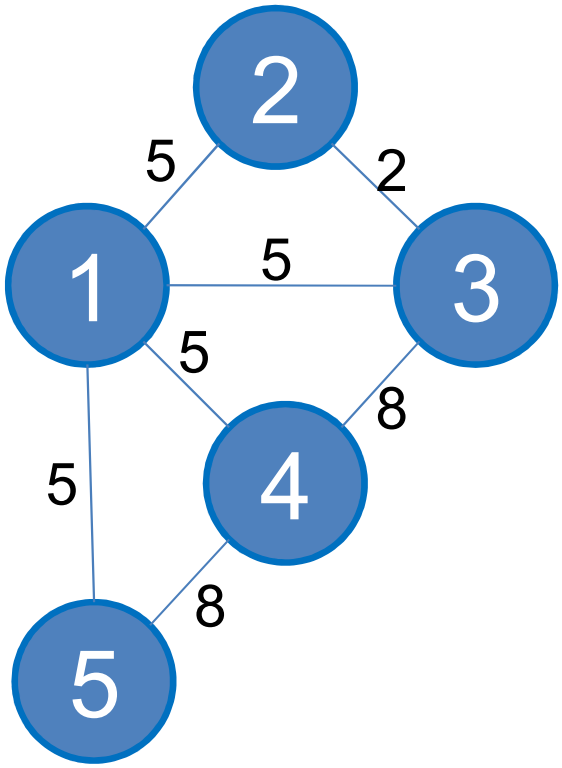


CP4 Section 4.3

MINIMUM SPANNING TREE

How to Solve This?

Given this graph, select some edges s.t
the graph is connected
but with minimal total weight!



Answer: MST!

Spanning Tree & MST

Given a **connected undirected** graph G,
select E $\in$ G such that a tree is formed
and this tree **spans** (covers) all V $\in$ G!

- No cycles or loops are formed!

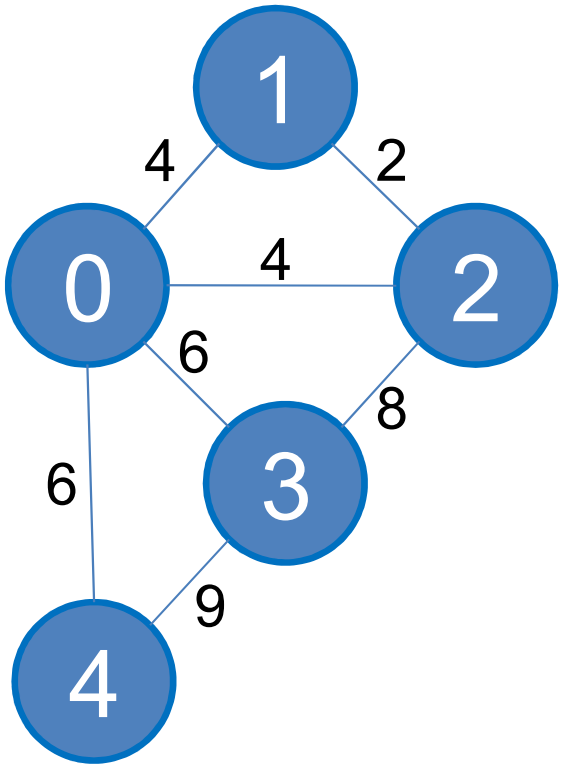
There can be **several** spanning trees in G

- The one where total cost is minimum
  is called the **Minimum** Spanning Tree (**MST**)

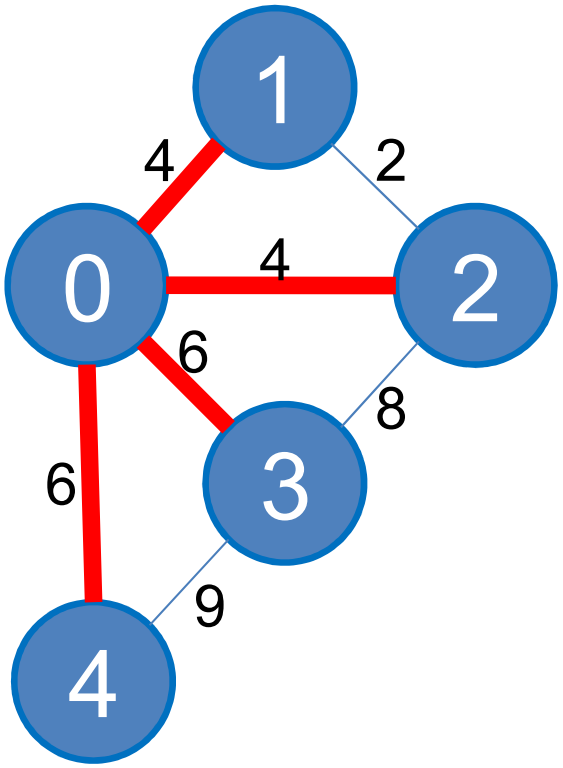
Example: UVa 908 (Re-connecting Computer Sites)

Example

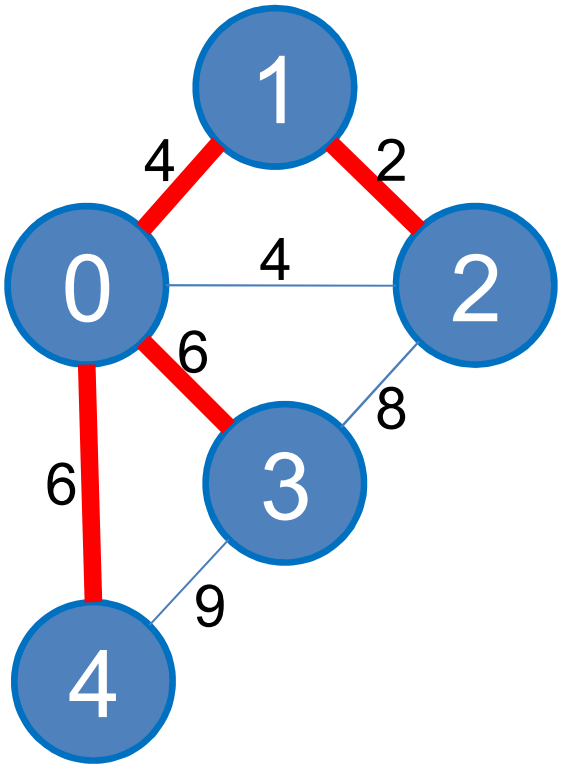
The Original Graph



A Spanning Tree
Cost: 4+4+6+6 = 20



An MST
Cost: 4+6+6+2 = 18



Algorithms for Finding MST

A. Prim's (Greedy Algorithm)
   - At every iteration, choose an edge with minimum cost that does not form a cycle
     - "grows" an MST from a root
B. Kruskal's (also Greedy Algorithm)
   - Repeatedly finds edges with minimum costs that does not form a cycle
     - forms an MST by connecting forests

Which one is easier to code?

Kruskal's Algorithm

Actually both, but I have personal preference: Kruskal's, see http://visualgo.net/en/mst

```
sort edges by increasing weight O(E log E)
while there are unprocessed edges left O(E)
  pick an edge e with minimum cost
  if adding e to MST does not form a cycle
    add e to MST
```

1. Simply store the edges in an array of Edges (EdgeList) and sort them, or use Priority Queue
2. Test for cycles using Disjoint Sets (Union Find) DS

Kruskal's Algorithm

https://github.com/stevenhalim/cpbook-code/blob/master/ch4/mst/kruskal.cpp

Uses: https://github.com/stevenhalim/cpbook-code/blob/master/ch2/ourown/unionfind_ds.cpp

```
sort(EL.begin(), EL.end());                          // sort by w, O(E log E)
// note: std::tuple has built-in comparison function

int mst_cost = 0, num_taken = 0;                     // no edge has been taken
UnionFind UF(V);                                     // all V are disjoint sets
// note: the runtime cost of UFDS is very light
for (int i = 0; i < E; ++i) {                        // up to O(E)
  auto [w, u, v] = EL[i];                            // C++17 style
  if (UF.isSameSet(u, v)) continue;                  // already in the same CC
  mst_cost += w;                                     // add w of this edge
  UF.unionSet(u, v);                                 // link them
  ++num_taken;                                       // 1 more edge is taken
  if (num_taken == V-1) break;                       // optimization
}
// note: the number of disjoint sets must eventually be 1 for a valid MST
printf("MST cost = %d (Kruskal's)\n", mst_cost);
```

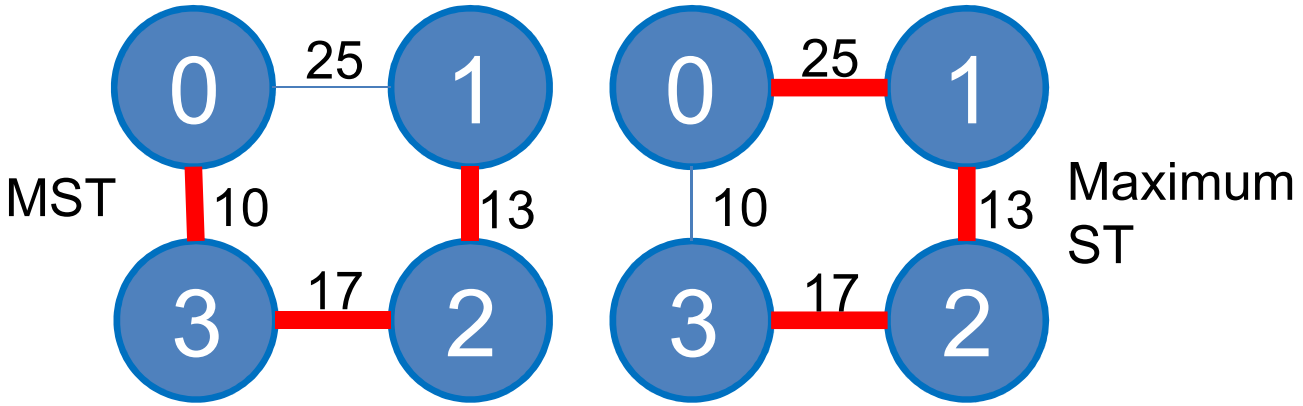
MST Variants (1)

Variants of basic MST problem are interesting!

Maximum ST

(LA 4110)

- Instead of minimum, we want the maximum ST
- Solution: Reverse the sort order in Kruskal's algorithm!



UVa 1234/LA 4110 – RACING

Singapore ICPC Regional 2007

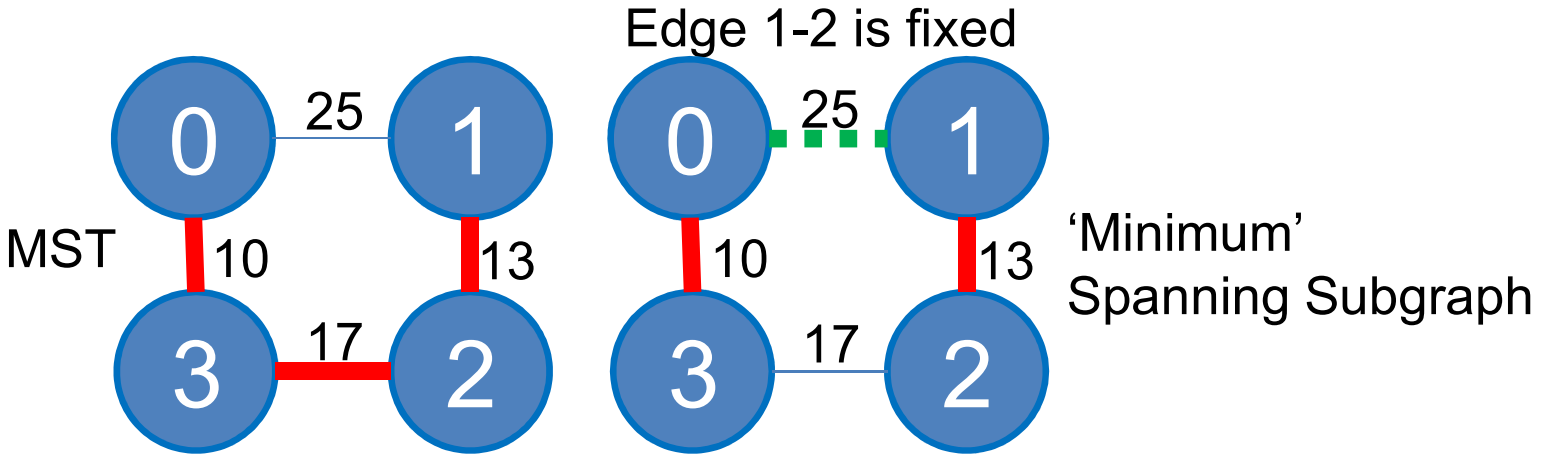
- Given a complex problem statement…
  - ~~Let's read the problem statement…~~
- Problem in essence:
  - Find the **Maximum** Spanning Tree
    by removing the lightest edges
- Solution:
  - Solvable using Kruskal's Algorithm with minor tweak:
    Reverse the sorted order

MST Variants (2)

‘Minimum’ Spanning Subgraph

(UVa: 10147, 10397)

- Some edges are **fixed**
- We need to continue building the Spanning Subgraph
- Solution: Use Kruskal’s algorithm to continue!

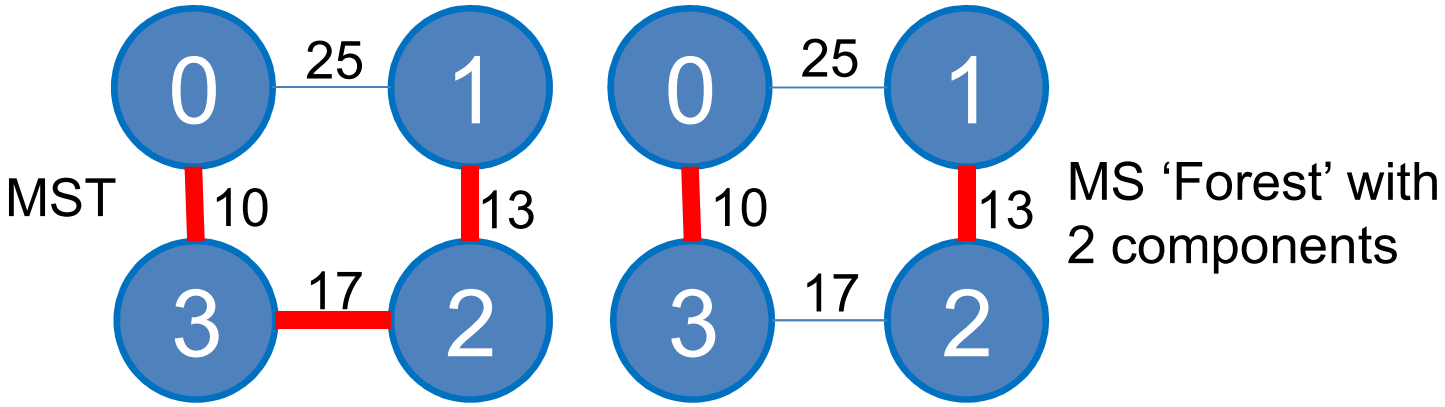


MST Variants (3)

Minimum 'Spanning Forest'

(UVa: 1216, 10369)

- Form a forest of k subtrees in the least cost way
- The desired number of components (k) is given
- Solution: Use Kruskal's algorithm again,
  stop when the number of connected component = k

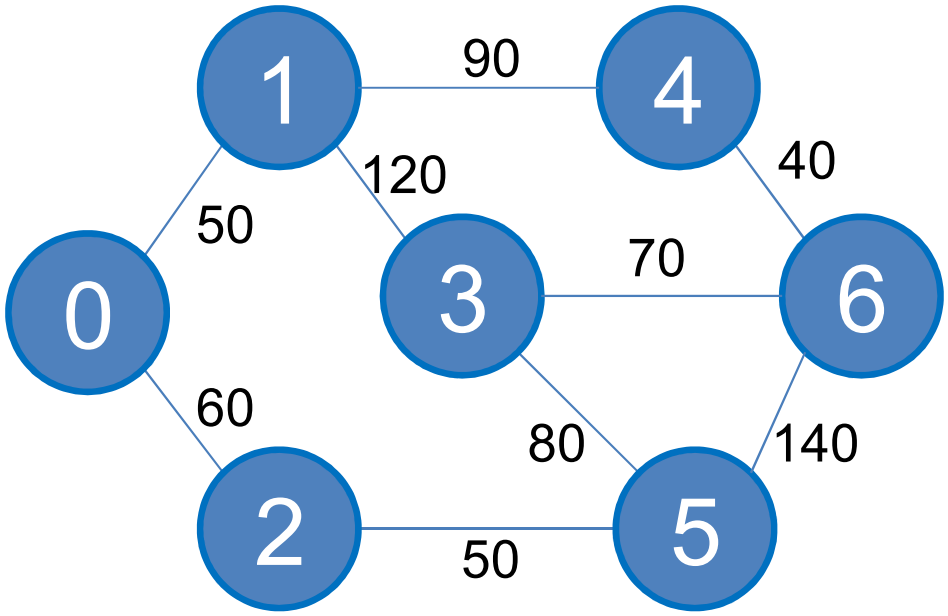


MST Variants (4) – (1)

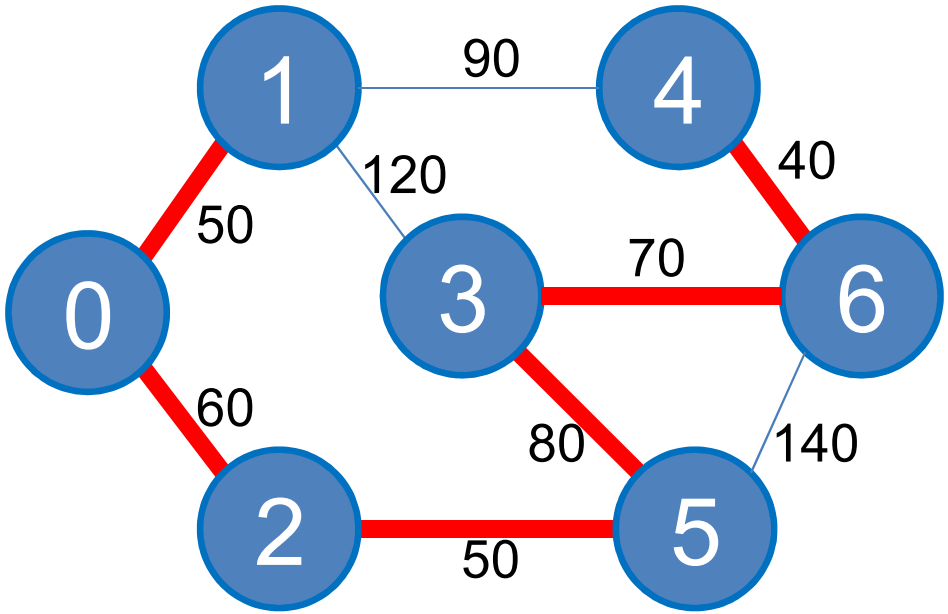
The MiniMax Path Problem

- Find a path from vertex a to vertex b that **mini**mizes the **max**imum edge weight passed along the path
- The reverse: MaxiMin
- $O(V^3)$ Floyd Warshall's solution exists, but we can do better

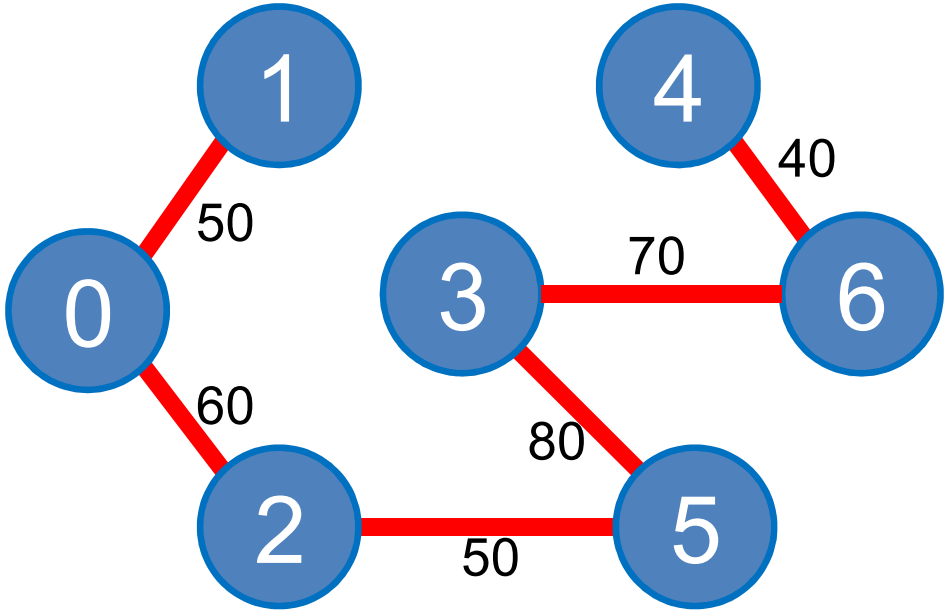
MST Variants (4) – (2)



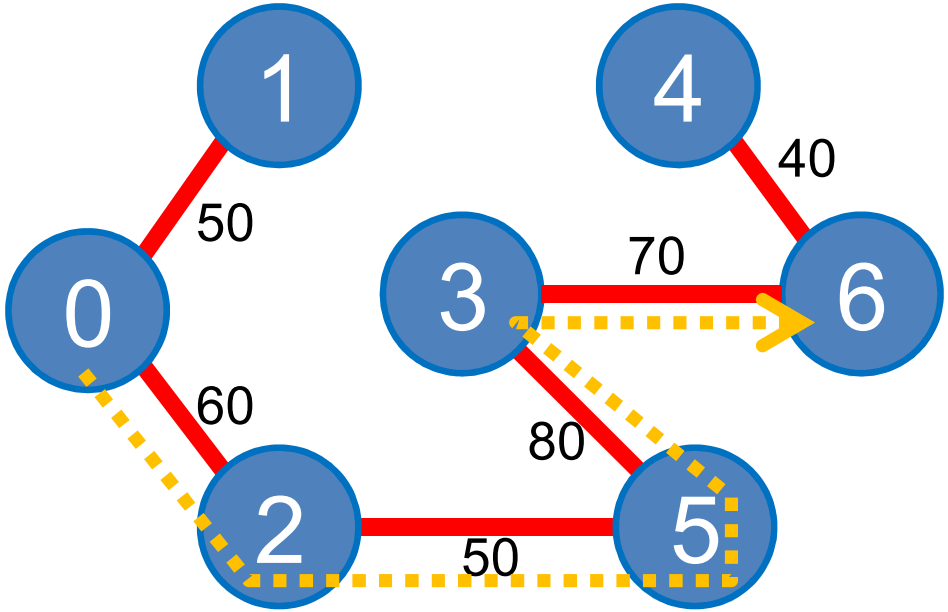
MST Variants (4) – (3)



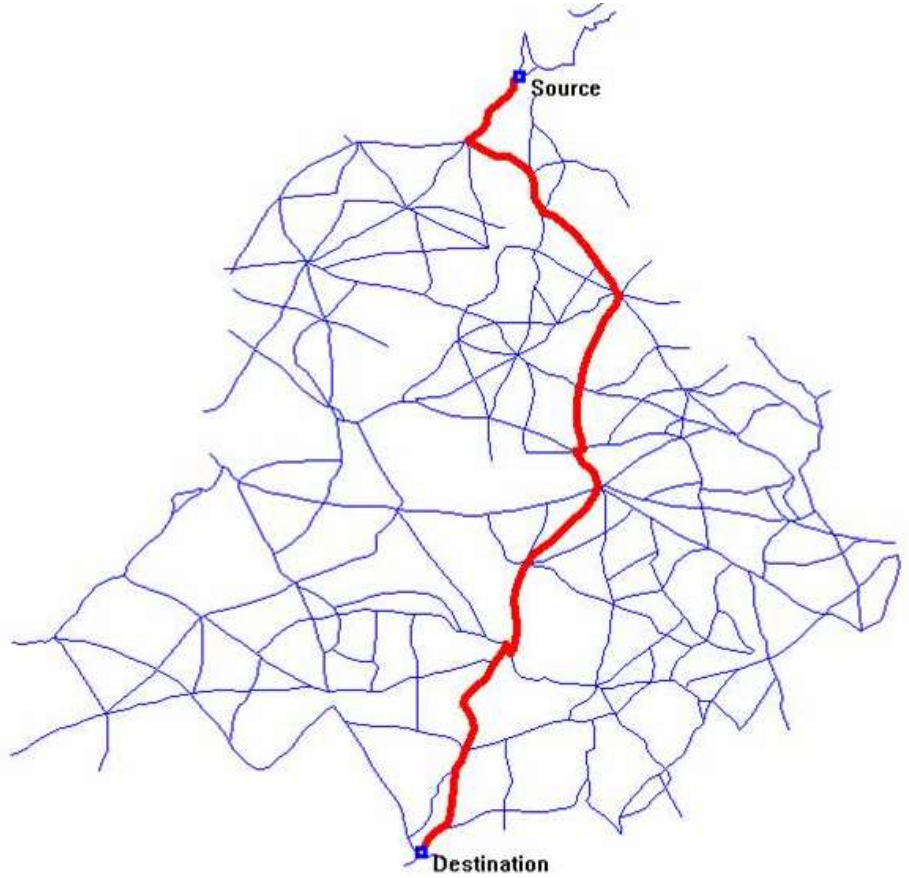
MST Variants (4) – (3)



MST Variants (4) – (4)



O(E log V) Kruskal's + O(V) DFS/BFS on the MST



BFS (unweighted/constant weight)

Dijkstra's (non -ve cycle)

~~Bellman Ford's (may have –ve cycle)~~

Floyd Warshall's (all-pairs)

SHORTEST PATHS

https://visualgo.net/en/sssp

BFS for **Special Case** SSSP

SSSP is a classical problem in Graph theory:

- Find shortest paths from **one source** to the rest^

Example of a special case: UVa 336 (A Node Too Far)

- Abridged Problem Description:
  - Given an **unweighted** & undirected Graph,
    a starting vertex ***v***, and an integer **TTL**
  - Check how many vertices are un-reachable from ***v***
    or has distance > **TTL** from ***v***
    - i.e. length(**shortest_path**(v, vertex)) > TTL

For SSSP on Weighted Graph but without Negative Weight Cycle

DIJKSTRA's

https://visualgo.net/en/sssp

Single-Source Shortest Paths (1)

If the graph is **unweighted**, we can use BFS

- But what if the graph is **weighted**?
- Assuming that there is no negative weight cycle…

Example: UVa 341 (Non Stop Travel)

Solution: O((V+E) log V) Dijkstra's algorithm

- A Greedy Algorithm
- Use Priority Queue

Dijkstra's Algorithm

https://github.com/stevenhalim/cpbook-code/blob/master/ch4/sssp/dijkstra.cpp

```
priority_queue<ii, vector<ii>, greater<ii>> pq; pq.push({0, s});

 // sort the pairs by non-decreasing distance from s
 while (!pq.empty()) {                              // main loop
   auto [d, u] = pq.top(); pq.pop();                // shortest unvisited u
   if (d > dist[u]) continue;                       // a very important check
   for (auto &[v, w] : AL[u]) {                     // all edges from u
     if (dist[u]+w >= dist[v]) continue;            // not improving, skip
     dist[v] = dist[u]+w;                           // relax operation
     pq.push({dist[v], v});                         // enqueue better pair
   }
 }
```

For All-Pairs Shortest Paths

FLOYD WARSHALL's

UVa 11463 – Commandos (1)

Al-Khawarizmi, Malaysia National Contest 2008

Given:

- A table that stores the amount of minutes to travel between buildings (there are **at most 100** buildings)
- 2 special buildings: startB and endB
- K soldiers to bomb all the K buildings in this mission
- Each of them start at the same time from startB, choose one building B that has not been bombed by other soldier (bombing time negligible), and then gather in (destroyed) building endB

What is the minimum time to complete the mission?

UVa 11463 – Commandos (2)

Al-Khawarizmi, Malaysia National Contest 2008

How long do you need to solve this problem?

Solution:

- The answer is determined by the shortest path from the starting building, detonate the **furthest building i**, and the shortest path from that furthest building i to the end building, i.e. max(dist[start][i] + dist[i][end]) $\forall i \in V$

But how to compute **sp** for **many** pairs of vertices?

UVa 11463 – Commandos (3)

Al-Khawarizmi, Malaysia National Contest 2008

This problem is called: All-Pairs Shortest Paths

Two options to solve this:

- Call SSSP algorithms multiple times
  - Dijkstra $O(V * (V+E) * \log V)$, if $E = V^2 \rightarrow O(V^3 \log V)$
  - Bellman Ford $O(V * V * E)$, if $E = V^2 \rightarrow O(V^4)$
  - Slow to code
- Use Floyd Warshall, a clever **DP** algorithm
  - $O(V^3)$ algorithm
  - Very easy to code!
  - In this problem, V is ≤ 400, so Floyd Warshall is feasible

Floyd Warshall – Template

$O(V^3)$ since we have three nested loops!

We must use adjacency matrix: `AM[MV][MV];`

- So that weight of edge(i, j) can be accessed in O(1)

```
for (int k = 0; k < V; ++k)
  for (int i = 0; i < V; ++i)
    for (int j = 0; j < V; ++j)
      AM[i][j] = min(AM[i][j], AM[i][k] + AM[k][j]);
```

See more explanation of this 4-liner DP in CP4

Remarks About SP Problems

- The hardest part is not the SP algorithm(s), but the **graph modeling**
- Single-Source AND Single-destination? (SSSDSP)
- Single-Destination SP? (SDSP)
- Multi-sources? (MSSP)
- Shortest Path Reconstruction
- 0-1 weighted SSSP?
- SSSP on small graph only?

CP4 Section 4.6

DAG

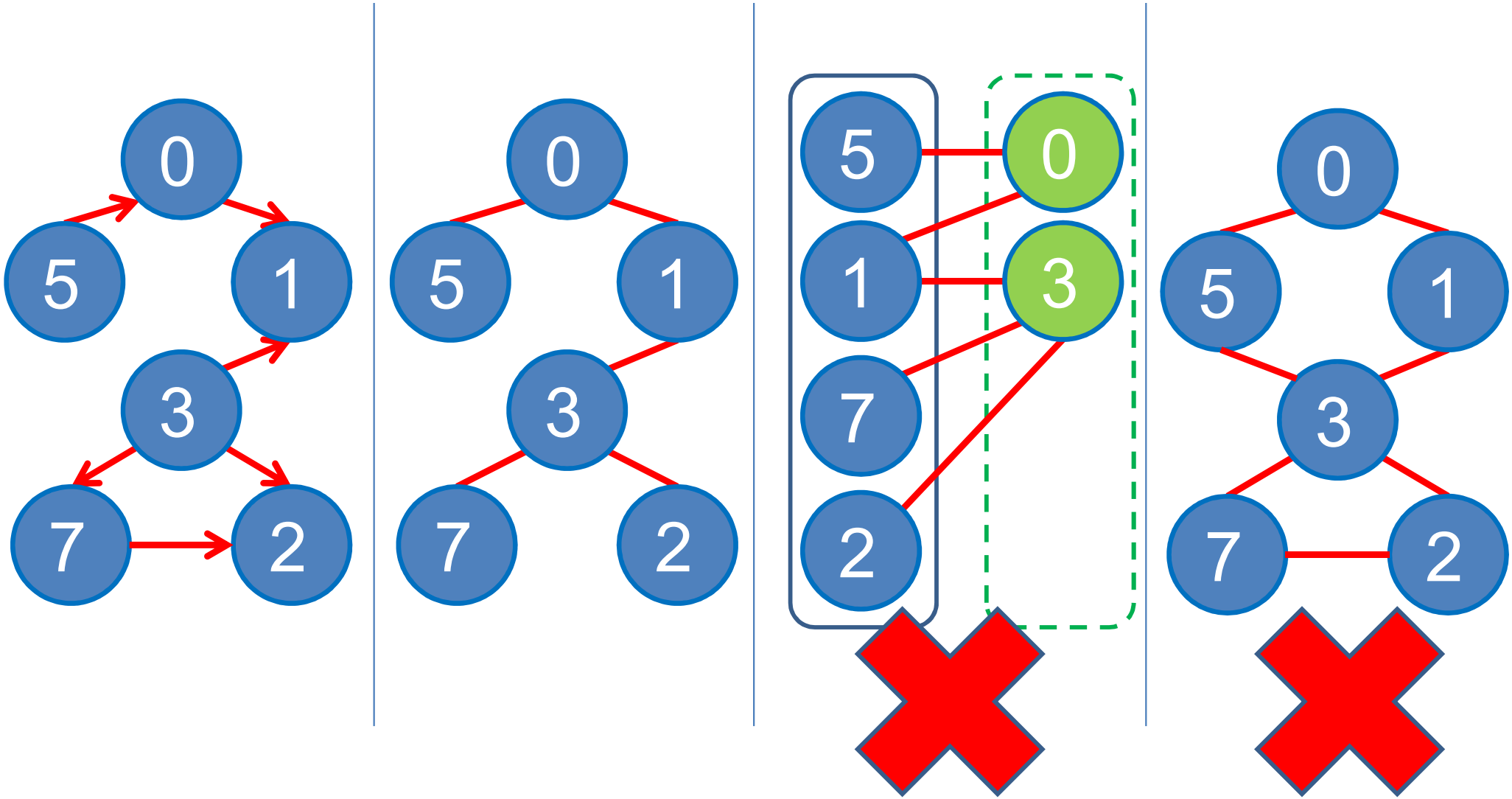
Tree

~~Bipartite~~

~~Eulerian~~

SPECIAL GRAPHS

The Special Graphs



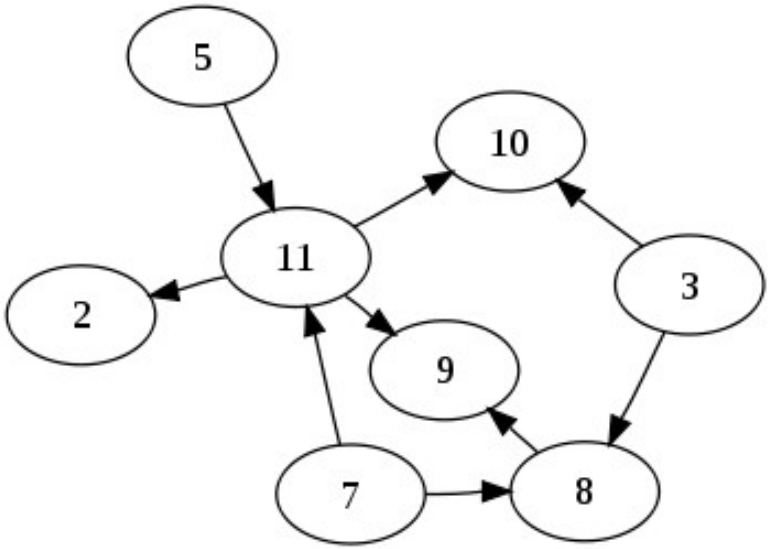
Special Graphs in Contest

There are a few special graphs that can appear in contest:

- Tree, keywords: connected, E = V-1, unique path!
- Directed Acyclic Graph, keywords: directed, no cycle/acylic
- ~~Bipartite, keywords: 2 sets, no edges within set!~~
- ~~Eulerian Graph, keywords: must visit each edge once~~
- ~~There are a few others, but rare~~

Some classical 'hard' (or NP-hard/Complete) problems may have *faster solution* on these special graphs

- This allows the problem author to increase the **input size**!
  - Eliminates those who are not aware of the faster solution as solution for general graph is slower (TLE) or harder to code (slower to get AC)…



DIRECTED ACYCLIC GRAPH (DAG)

https://visualgo.net/en/sssp

Single-Source Shortest Paths
in DAG

In general weighted graph

- This is O((V+E) log V) using Dijkstra's
  or O(VE) using Bellman Ford's

In DAG

- The fact that there is no cycle simplifies this problem substantially!
  - Simply "relax" vertices according to topological order!
  - This ensure shortest paths are computed correctly!
  - One topological sort can be found in O(V+E)

https://visualgo.net/en/sssp

(Single-Source) Longest Paths in DAG

In general weighted graph

- Determining whether a graph has a longest (simple) paths with at least K edge(s) is an **NP Complete** (read: EXTREMELY HARD) problem

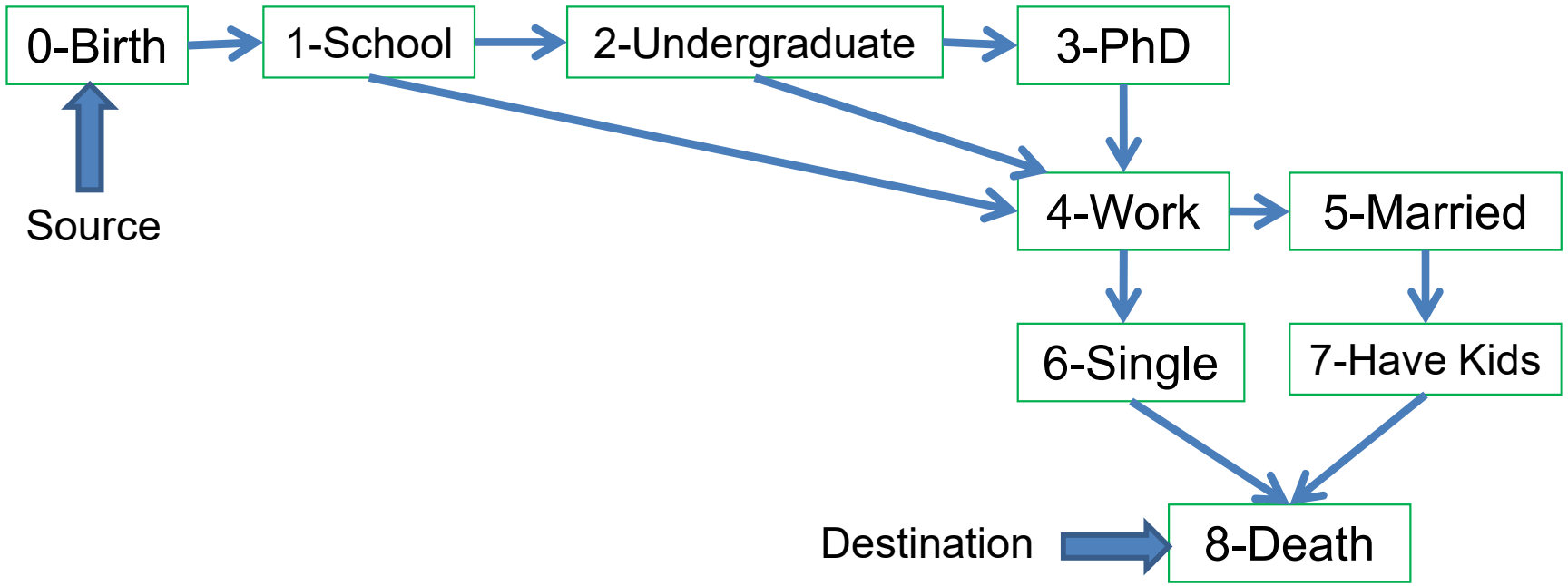
In DAG

- The solution is the same as shortest paths in DAG
- Just that we have tweak the relax operator
  (or alternatively, negate all edge weight in DAG)

Counting Paths on DAG

Given some real-life time line (obviously a DAG, Q: Why?)

- How many different possible lives that you can live (from birth/vertex 0 to death/vertex 8)?

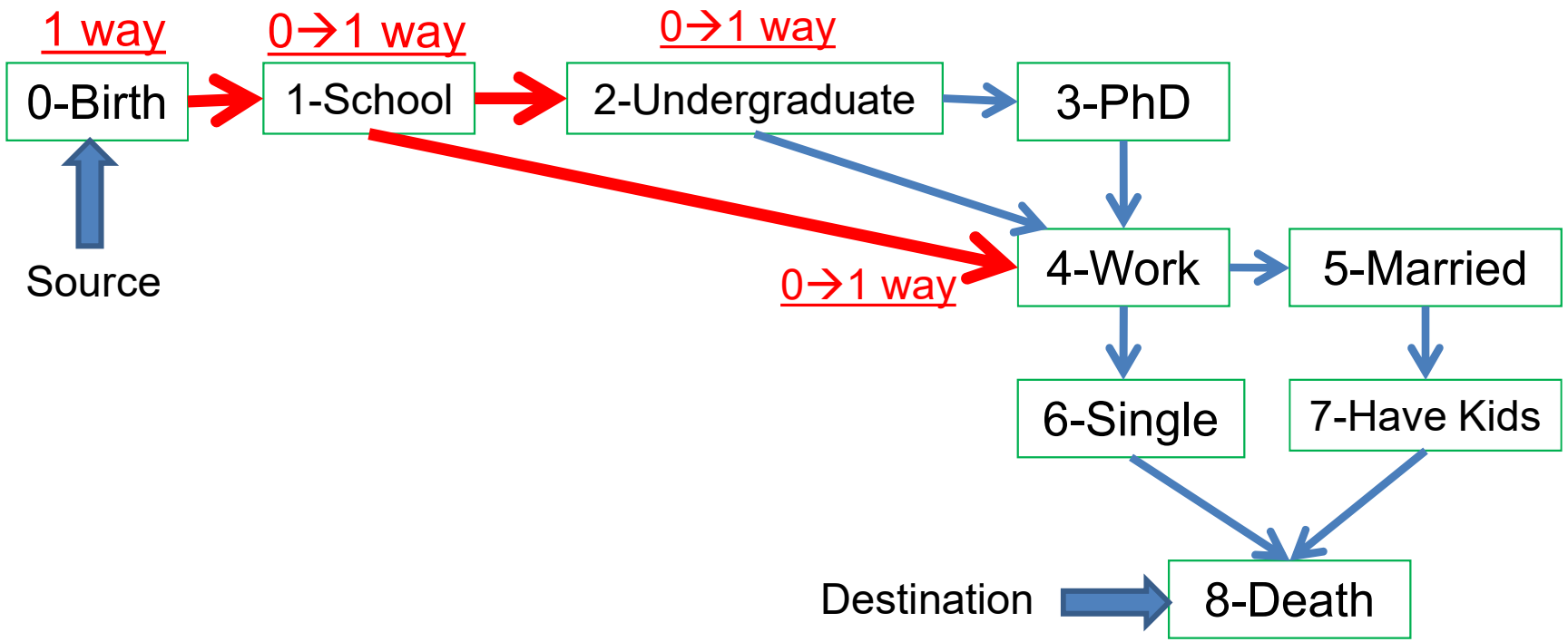


Answer = 6

Toposort Solution (1)

Find the toposort first: {0, 1, 2, 3, 4, 6, 5, 7, 8}

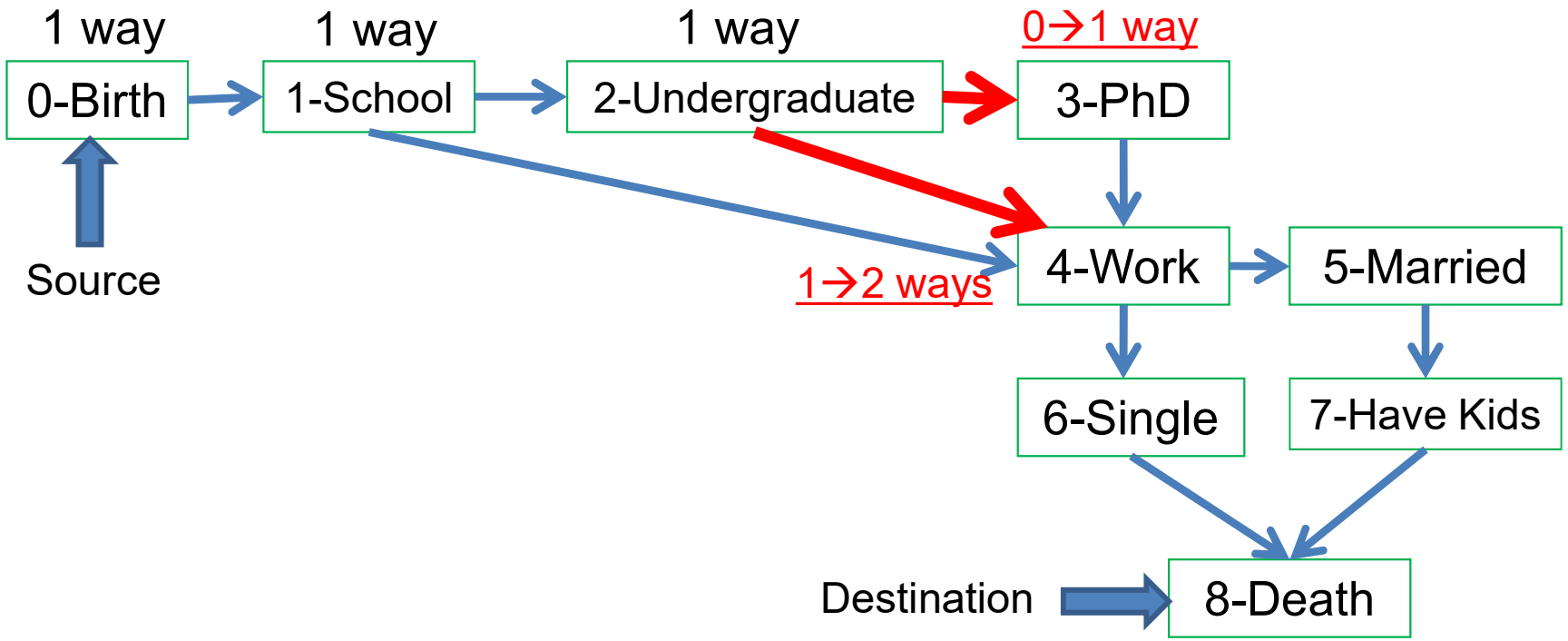
- numPaths[0] = 1, propagate to vertex 1, and then 2, 4



Toposort Solution (2)

Find the toposort first: {0, 1, 2, 3, 4, 6, 5, 7, 8}

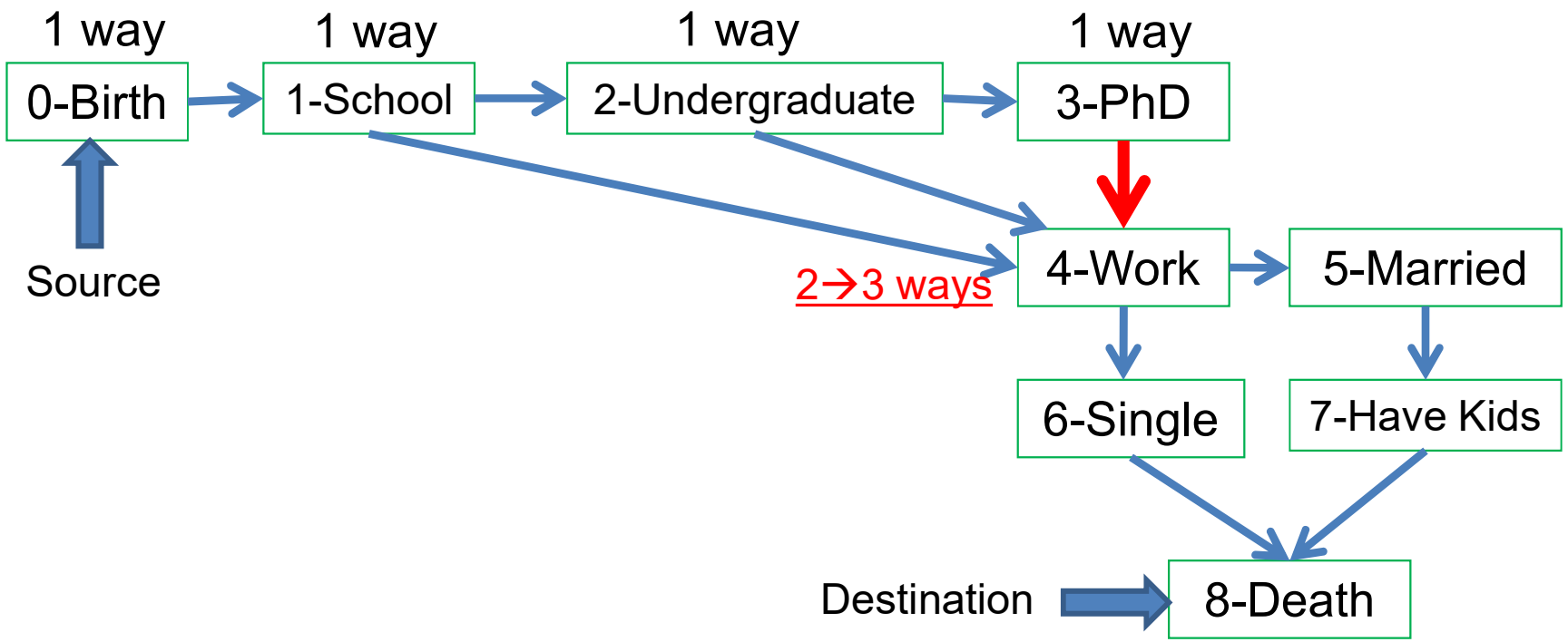
- numPaths[2] = 1, propagate to vertex 3 and 4



Toposort Solution (3)

Find the toposort first: {0, 1, 2, 3, 4, 6, 5, 7, 8}

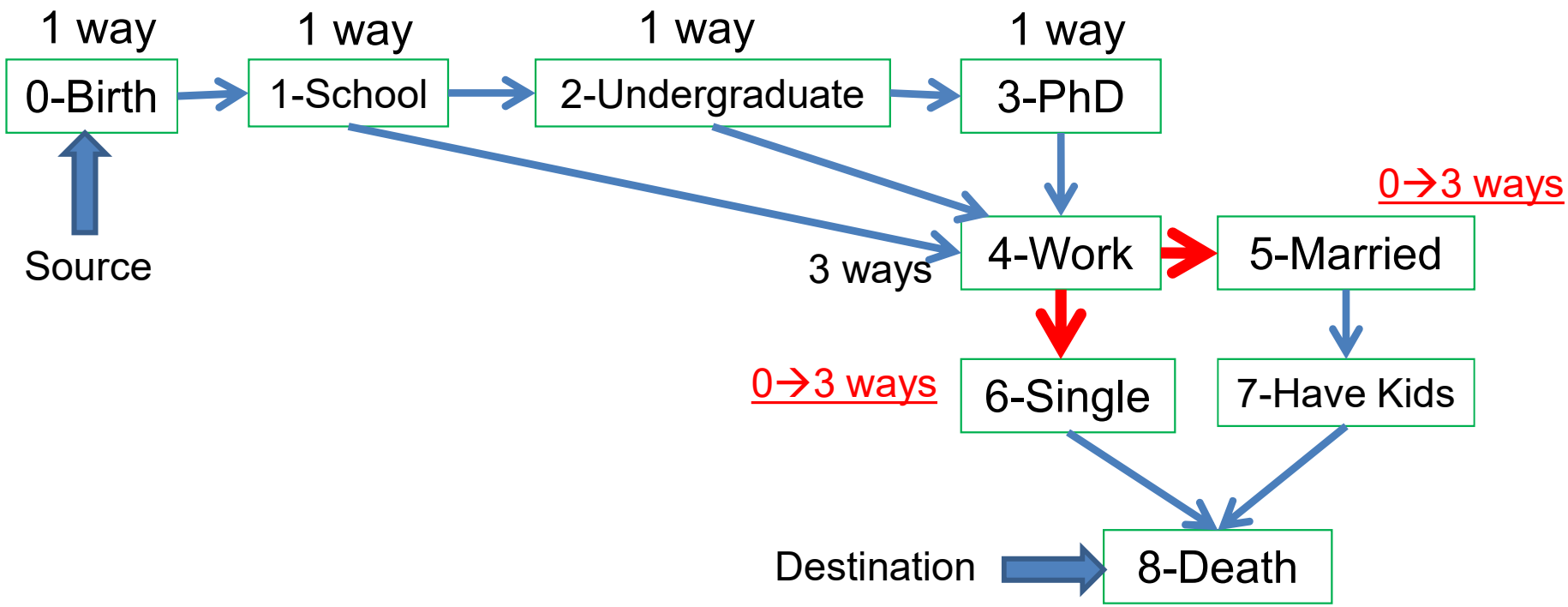
- numPaths[3] = 1, propagate to vertex 4



Toposort Solution (4)

Find the toposort first: {0, 1, 2, 3, 4, 6, 5, 7, 8}

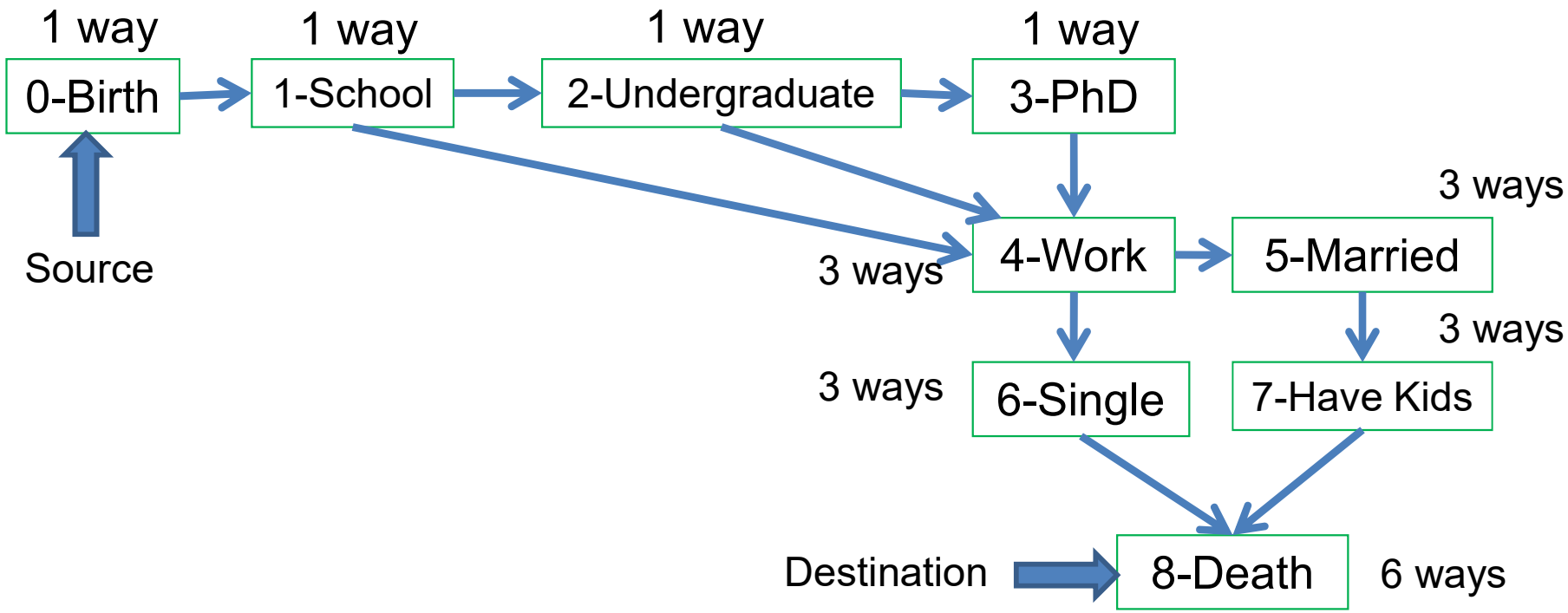
- numPaths[4] = 3, propagate to vertex 5 and 6



Toposort Solution (5)

Find the toposort first: {0, 1, 2, 3, 4, 6, 5, 7, 8}

- >> >> Fast forward…, this is the final state



TREE

Tree

Tree is a special graph with these characteristics:

- Connected
- Has V vertices and exactly E = V-1 edges
- Has no cycle / acyclic
- Has one unique path between two vertices
- Every tree is a bipartite graph
- Sometimes, it has one special vertex called “root” (rooted tree): Root has no parent
- A vertex in n-ary tree has either {0, 1,…,n} children
  - n = 2 is called binary tree (the most popular one)

(Binary) Tree Traversal

In general graph

- We need O(V+E) DFS or BFS

In Binary Tree

- Pre-order, In-order, Post-order
  - No speed up, but coding is a little bit simpler

SSSP and APSP Problems
on Weighted Tree

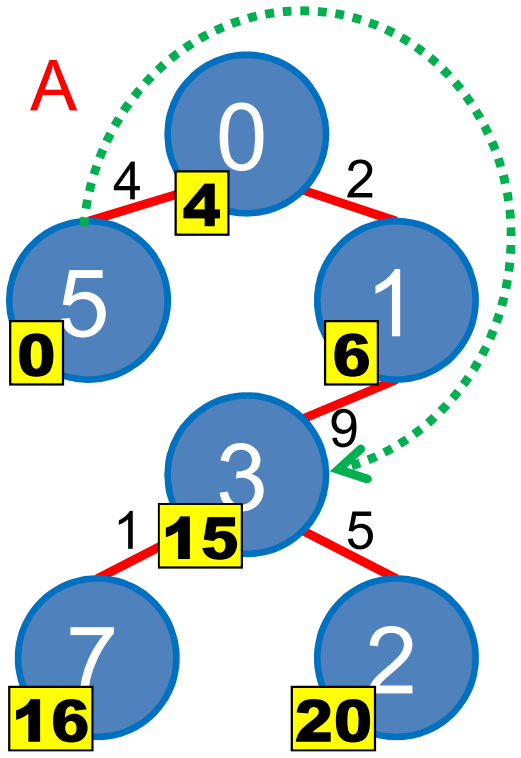
In general weighted graph

- SSSP problem: O((V+E) log V) Dijkstra's
  or O(VE) Bellman Ford's
- APSP problem: $O(V^3)$ Floyd Warshall's

In weighted tree

- SSSP problem: O(V+E = V+V = V) DFS or BFS
  - There is only 1 unique path between 2 vertices in tree
- APSP problem: simple V calls of DFS or BFS: $O(V^2)$
  - But can be made even faster using LCA… not discussed today

Tree Illustration



SSSP on Tree

Summary (1)

Today, we have gone through various well-known graph problems & algorithms

- Graph Data Structures: AM, AL, EL, Implicit
- Graph Traversal: DFS/BFS
- Minimum Spanning Tree: Kruskal's, ~~Prim's~~
- Shortest Paths: BFS, Dijkstra's, ~~Bellman Ford's~~, Floyd Warshall's
- Special Graphs: DAG, Tree, ~~Bipartite, Eulerian~~

Summary (2)

Note that just knowing these algorithm will not be too useful in contest setting…

You have to practice **using them**

- At least code each of the algorithms discussed today on a contest problem!

And you need to develop good **"graph modeling"** skills

Summary (3)

Graph problems appear several times in ICPC!

- Min 1, normally 2, can be 3 out of 10
- Master all known solutions for classical graph problems
- Or perhaps combined with DP/Greedy style

This can move your team nearer to top 10

Graph problems do appear in IOI (see IOI syllabus)

- Knowing (all?) these basic algorithms will surely help

References

- CP4, Chapter 4, parts of Chapter 9
- Introduction to Algorithms, Ch 22,23,24,25,26 (p643-698)
- Algorithm Design, Ch 3,4,6,7 (p337-450)
- Algorithms (Dasgupta et al), Ch 6 & Ch 7
- Algorithms (Sedgewick), Ch 33 & Ch 34
- Algorithms (Alsuwaiyel), Ch 16 & Ch 17
- Programming Challenges, p227-230, Ch 10
- http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=standardTemplateLibrary2
- Internet: TopCoder Max-Flow tutorial, UVa Live Archive, UVa main judge, Felix's blog, Suhendry's blog, Dhaka 2005 solutions, other Max Flow lecture notes, etc…