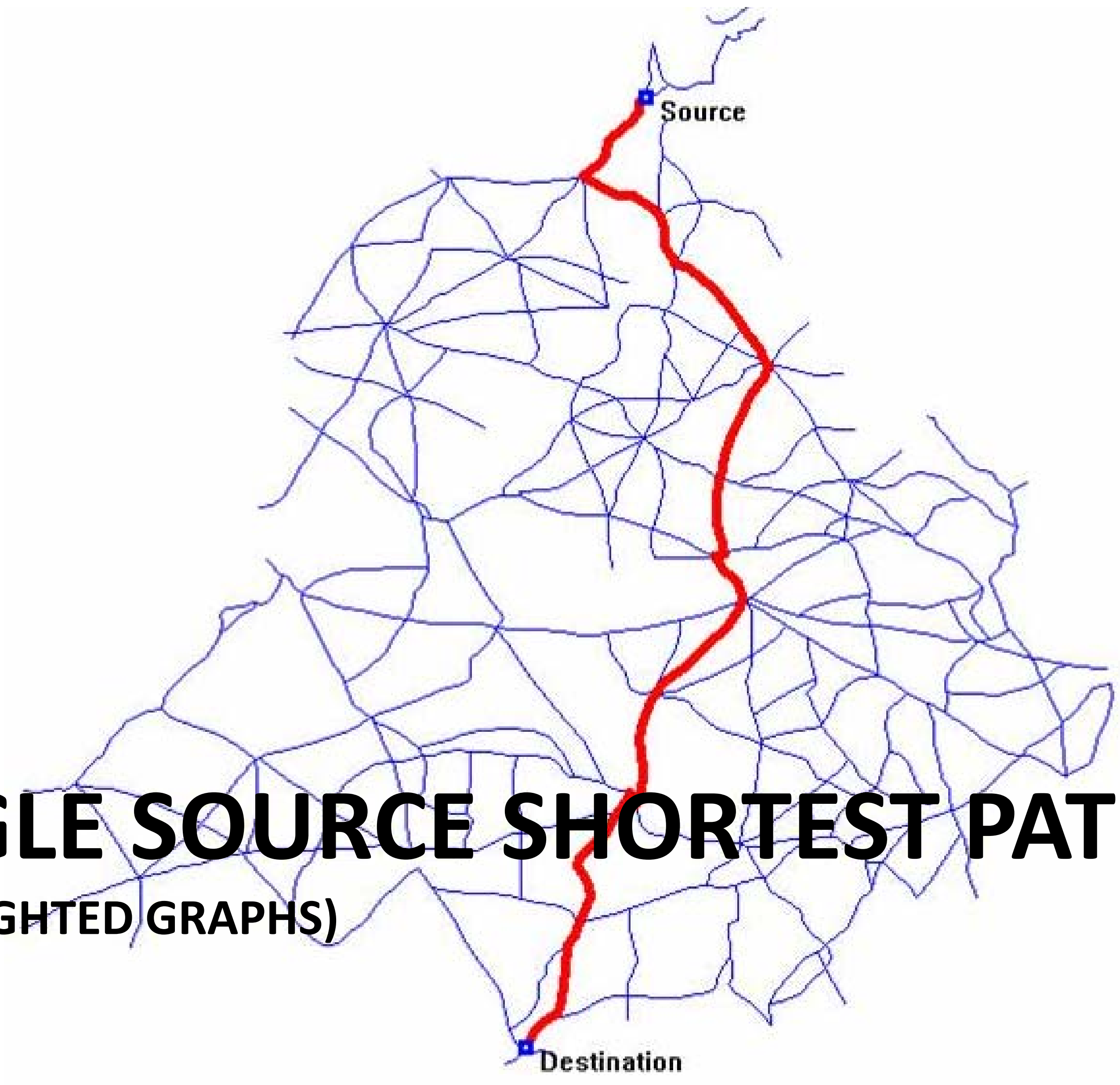
CS3233
Competitive Programming

Dr. Steven Halim

Week 06 – More Graph

Outline

- Mini Contest #5
  - Break
- Single-Source Shortest Paths (SSSP)
  - Dijkstra’s, Bellman Ford’s
- All-Pairs Shortest Paths (APSP)
  - Floyd Warshall’s + Variants (Transitive Closure/Maximin/Minimax)
- Special Graphs Part 1
  - Tree, Euler Graph, Directed Acyclic Graph
- Be a Problem Setter



SINGLE SOURCE SHORTEST PATHS

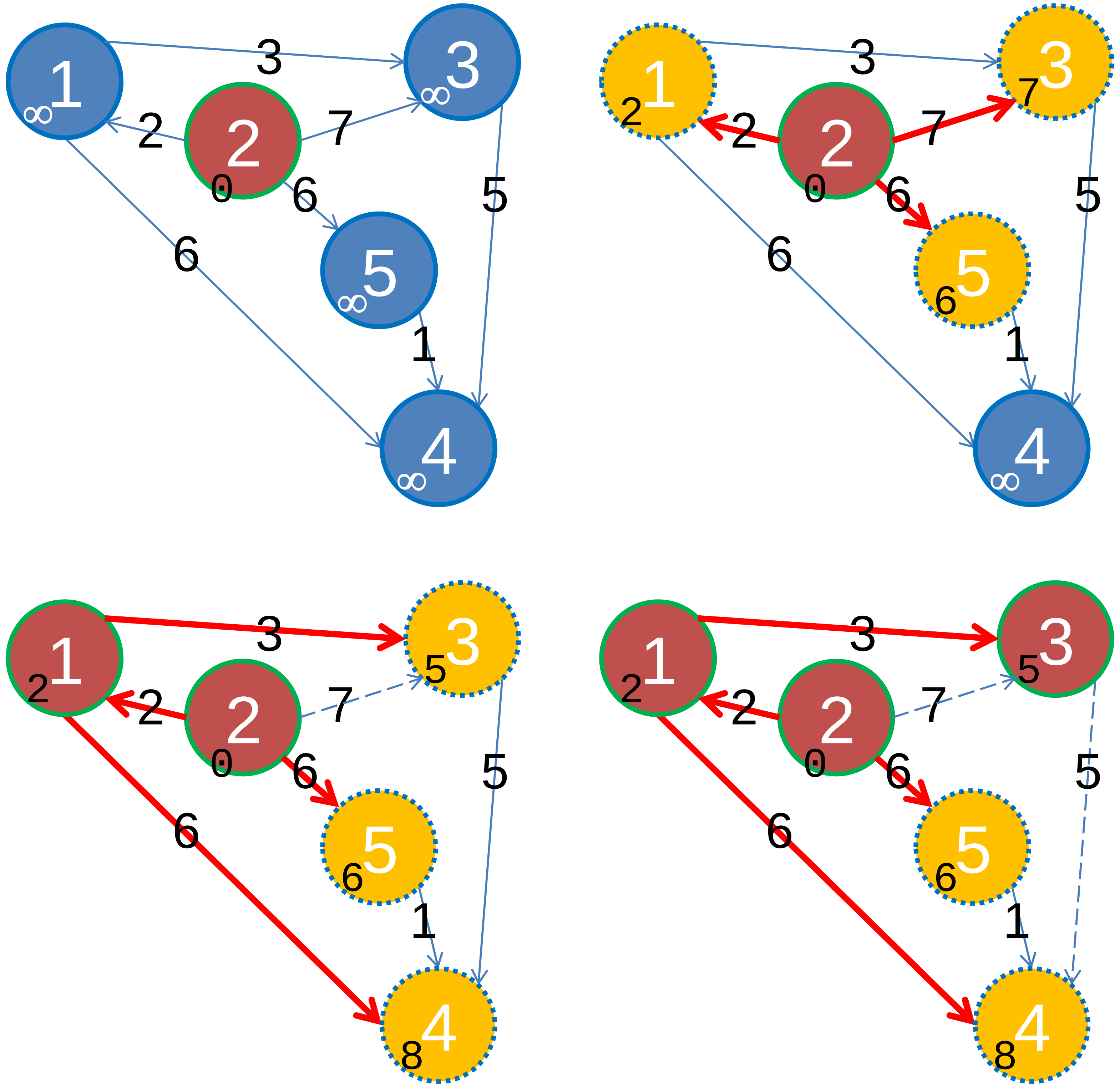
(ON WEIGHTED GRAPHS)

DIJKSTRA's

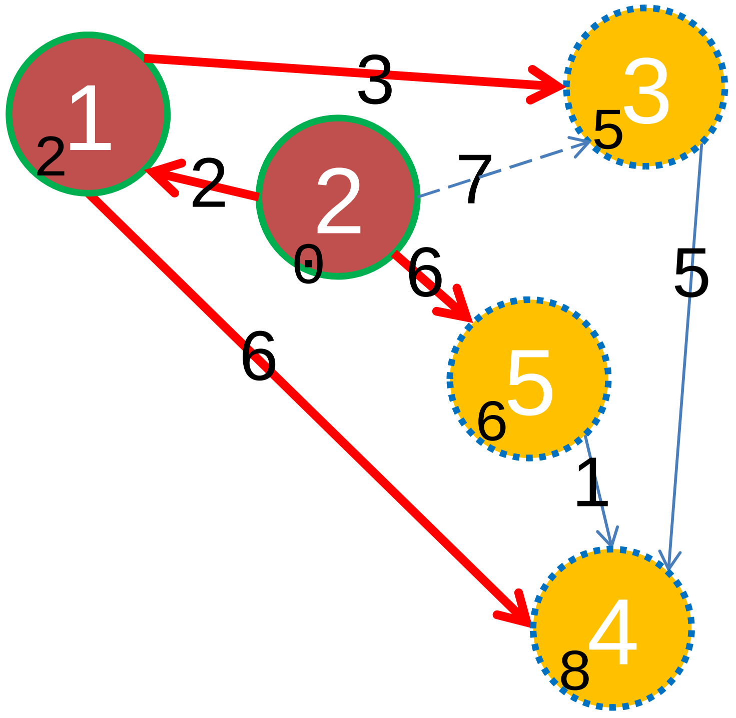
Single Source Shortest Paths (1)

- Classical problem in Graph theory:
  - Find shortest paths from **one source** to the rest^
- If the graph is **un weighted**, we can use BFS
  - But what if the graph is **weighted**?
- UVa: 341 (Non Stop Travel)
- Solution: Dijkstra O((V+E) log V)
  - A Greedy Algorithm
  - Use Priority Queue

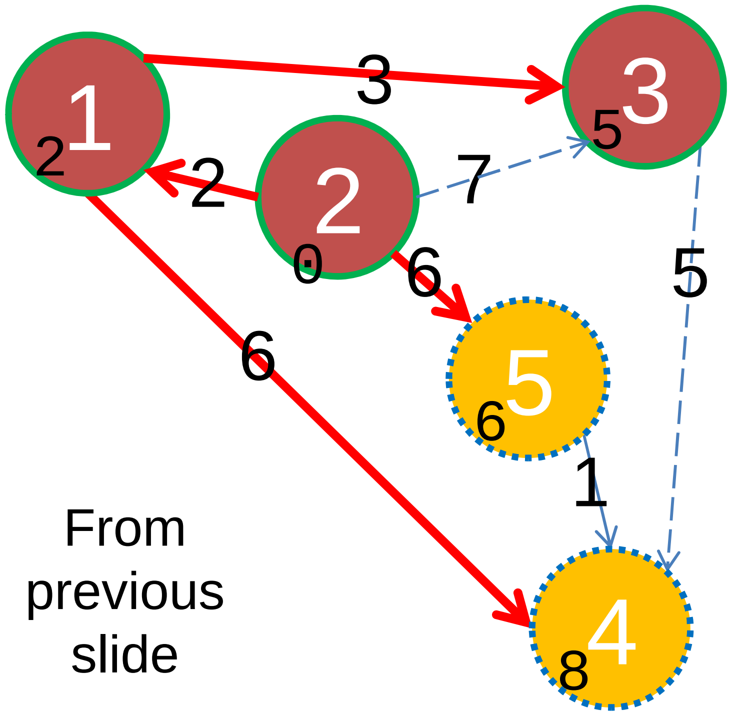
UVa 341 (1)

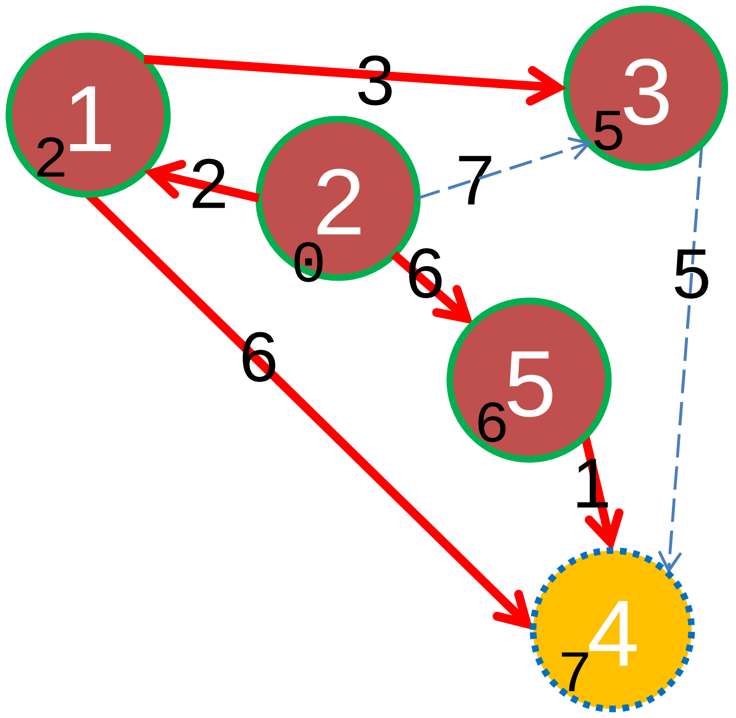


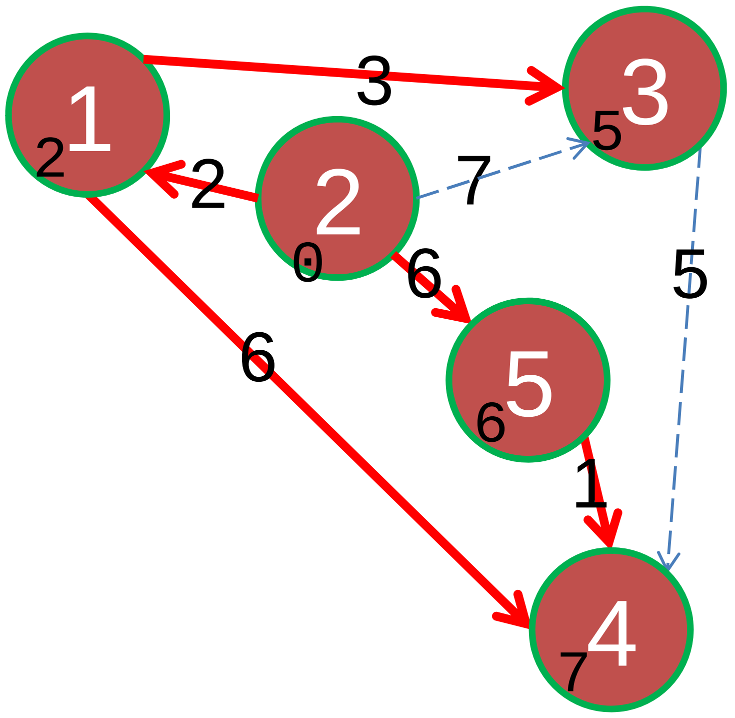
UVa 341 (2)



From previous slide







Dijkstra’s Algorithm (using STL)

```
vector<int> dist(V, INF); dist[s] = 0; // INF = 2B
priority_queue<ii, vector<ii>, greater<ii> > pq;
pq.push(ii(0, s)); // sort based on increasing distance
while (!pq.empty()) { // main loop
  ii top = pq.top(); pq.pop(); // greedy
  int d = top.first, u = top.second;
  if (d == dist[u]) {
    TRvii (AdjList[u], it) { // all outgoing edges from u
      int v = it->first, weight_u_v = it->second;
      if (dist[u] + weight_u_v < dist[v]) {
        dist[v] = dist[u] + weight_u_v; // relax
        pq.push(ii(dist[v], v)); // enqueue this neighbor
      }                          // regardless it is
    }                            // already in pq or not
  }
}
```

UVa 11492 – Babel (1)

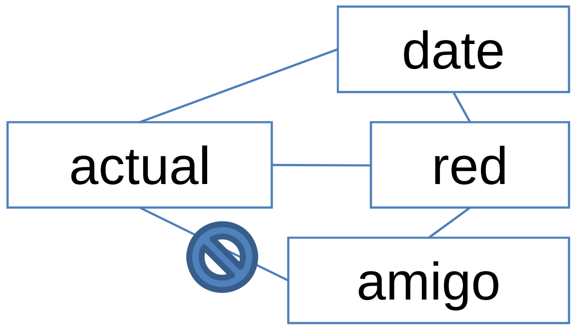
- Given start, finish language, and list of words that are common in two languages
  - e.g. the word “red” is a “color” in English, but it means “network” in Spanish
- Find chain of words that satisfy the lang links, 1st char rule, and min its num of total chars:
  - Lang links: w_1 (L_{start}/L_a), $w_2(L_a/L_c)$, …, w_i (L_x/L_{finish})
  - 1st char rule: $w_1[0]$!= $w_2[0]$
  - Total length of chars: $\text{len}(w_1) + \text{len}(w_2) + \ldots \text{len}(w_i)$

UVa 11492 – Babel (2)

- See example for clarity
  - Example (max 2000 words):
    - Start language: portugues, Finish language: frances
    - Words: ingles espanhol **red**, espanhol portugues **amigo**, frances ingles **date**, espanhol ingles **actual**
  - Infeasible 1: amigo date, length = 9
  - Infeasible 2: <u>**a**</u>migo <u>**a**</u>ctual date, length = 15
  - Not optimal: amigo red actual date, length = 18
  - Optimal: amigo red date, total length = 12

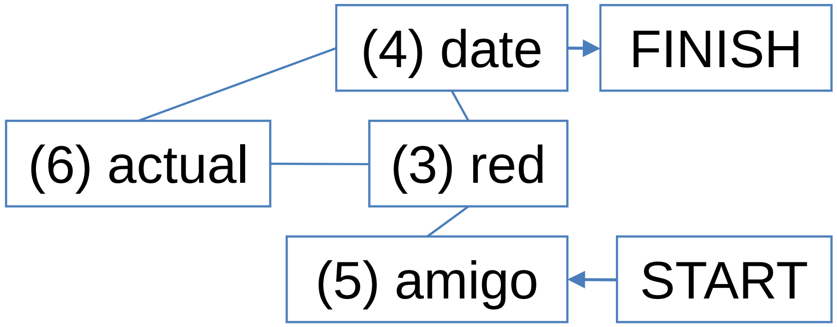
UVa 11492 – Babel (3)

- Where is the graph?
  - Each word is a vertex
  - Connect two vertices (words) with edge if they share common language & have different 1st char!
- But…
  - There can be many possible sources and destinations?
    - Many words of starting and finish language



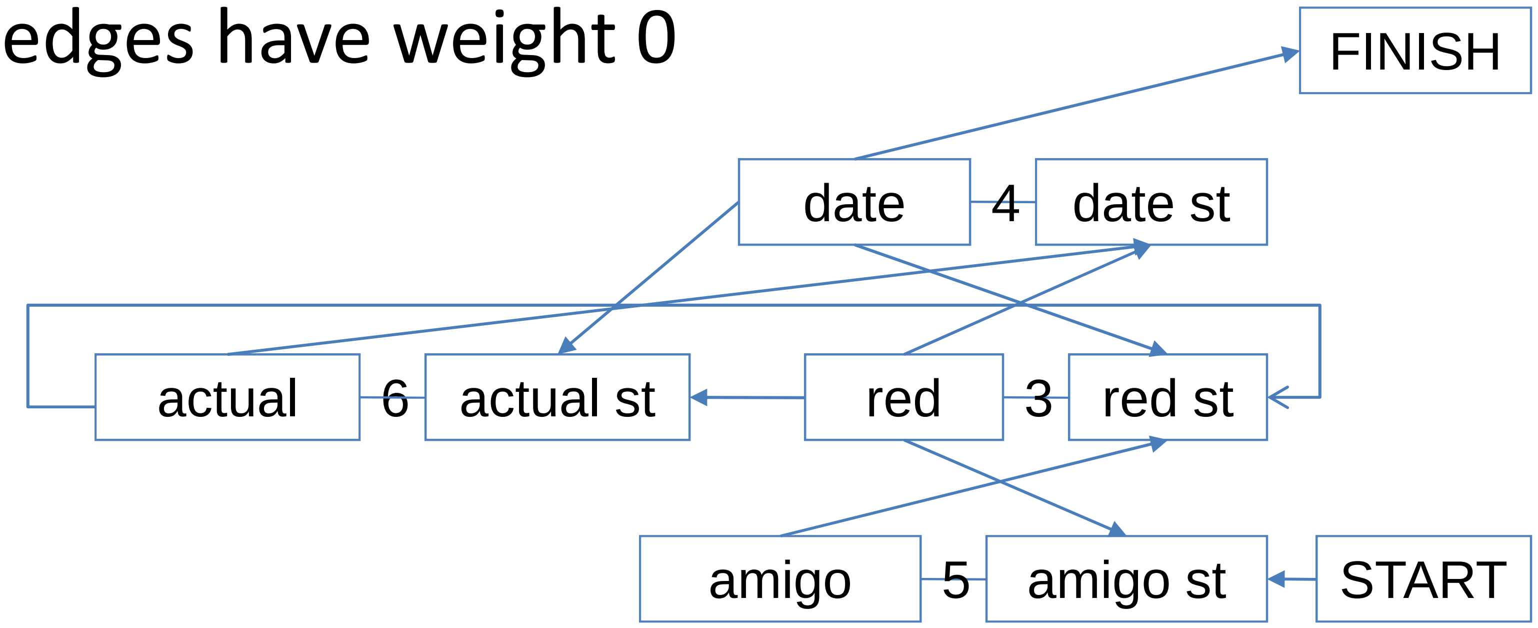
UVa 11492 – Babel (4)

- Where is the graph? – continued
  - Introduce "dummy vertices": START and FINISH
    - Connect "START" to all words that are inside the vocabulary of start language
    - Connect all words that are inside the vocabulary of finish language to "FINISH"
  - Seems like sssp, but…
    - There is no edge weight
    - But we have vertex weight of len(word)?



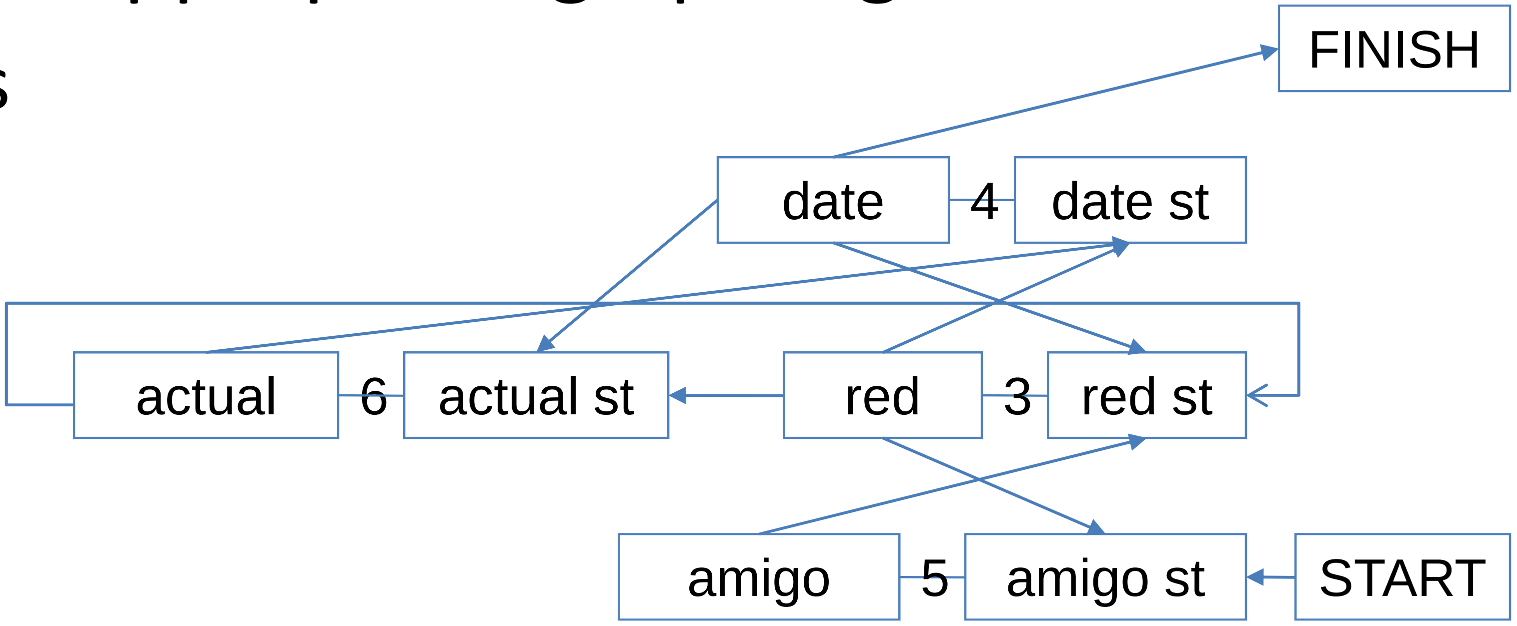
UVa 11492 – Babel (5)

- Where is the graph? – continued
  - Introduce "split vertices"
    - A vertex with weight X can be split into 2 vertices connected with edge of weight X
  - Normal edges have weight 0



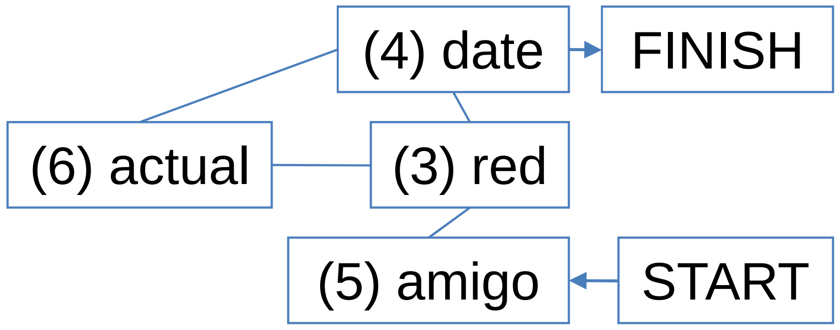
UVa 11492 – Babel (6)

- Now what is the graph problem?
  - sssp on weighted graph with non negative edge weight from START to FINISH
- What is the appropriate graph algorithm?
  - Dijkstra's



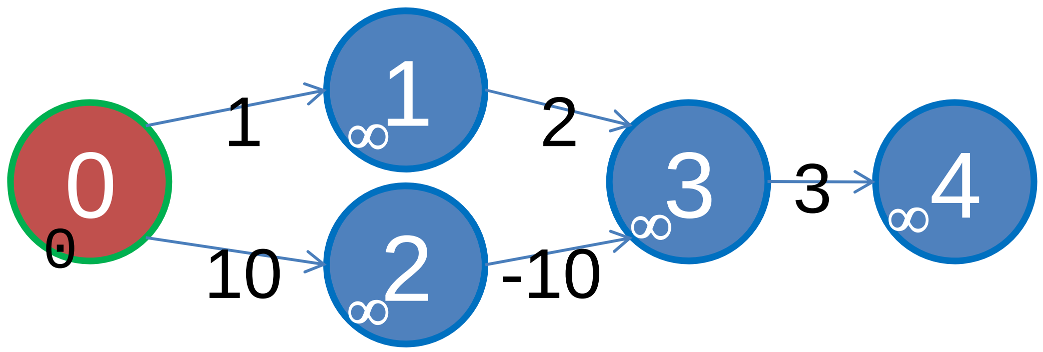
UVa 11492 – Babel (7)

- However…
  - We can simply use modified Dijkstra's here
    - Although all edges (v,u) have cost = 0, we can say that: D[u] = D[v] + 0 + wordlen[u]
    - Run Dijkstra's from START to FINISH



An Issue with Dijkstra’s

- If the graph has **negative edge weight**
  - Dijkstra’s fails
- Solution: Bellman Ford’s
  - Complexity^: O(V^3) or O(VE)



The same man who invented "Dynamic Programming" technique

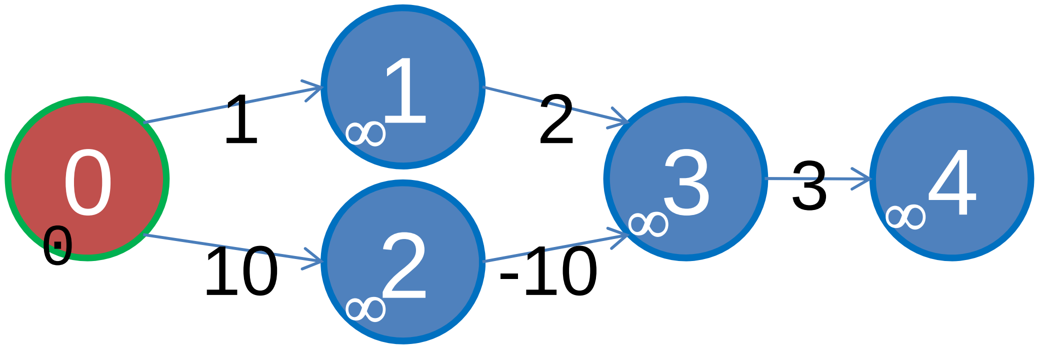
The same man who invented "Ford Fulkerson" method for Max Flow / Min Cut problem

BELLMAN FORD's

Bellman Ford's (using STL)

```
typedef vector<int> vi;
#define INF 2000000000 // (avoid overflow)

vi dist(V, INF); dist[s] = 0;
REP (i, 0, V - 2) // relax all E edges V-1 times, O(V)
  REP (u, 0, V - 1) // these two loops = O(E)
    TRvii (AdjList[u], v) // has edge and can be relaxed
      dist[v->first] = min(dist[v->first],
                           dist[u] + v->second);
```

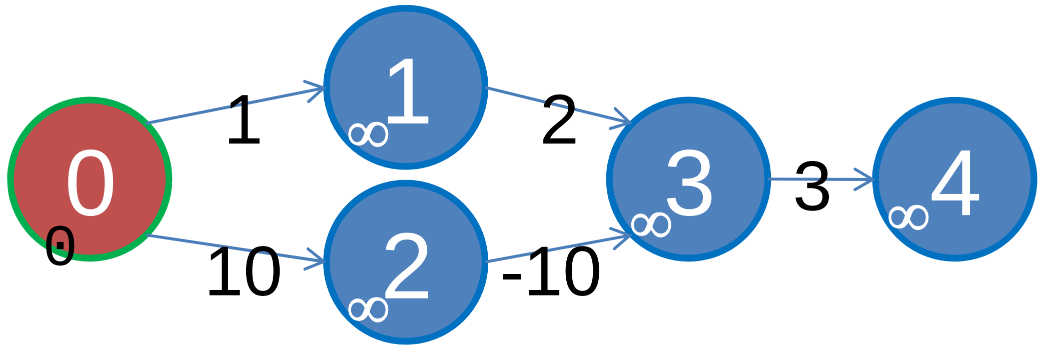


UVa 558 – Wormholes

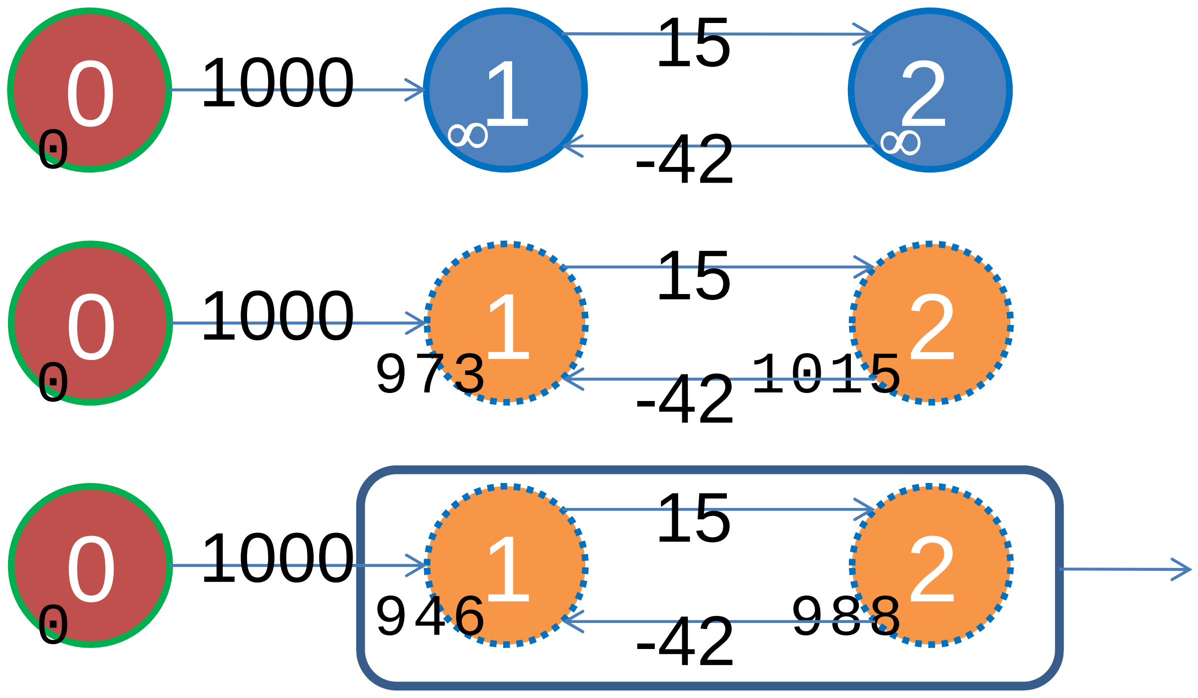
- Bellman Ford can also deal with **negative cycle**
- UVa: 558 (Wormholes)
  - There are X wormholes in star system.
  - If you enter wormhole i connected to wormhole j, you can teleport +x years to the future or -y years to the past.
  - There are up to 1000 wormholes and 2000 ‘time tunnels’.
  - Question: **Can you go back to the beginning of universe?**
    - Is there exist a negative cycle in this graph?
  - Bellman Ford can detect negative cycle
    (but this shortest paths problem cannot be solved)

Bellman Ford's (using STL)

```
vector<int> dist(V, INF); dist[s] = 0; // INF = 2B
REP (i, 0, V - 1) // relax all E edges V-1 times, O(V)
  REP (u, 0, V - 1) // these two loops = O(E)
    TRvii (AdjList[u], v) // has edge and can be relaxed
      dist[v->first] = min(dist[v->first],
                           dist[u] + v->second);
bool negative_cycle_exist = false;
REP (u, 0, V - 1) // one more pass to check
  TRvii (AdjList[u], v)
    if (dist[v->first] > dist[u] + v->second)
      negative_cycle_exist = true;
```



Visualization – UVa 558



Negative Cycle!

We should not be able to relax these edges anymore after V-1 steps of Bellman Ford's…

But in this example, we can… So this is a negative cycle!

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 sp from starting building, detonate **furthest building**, and sp from that furthest building to end building
    - max(dist[start][i] + dist[i][end]) for all i $\in$ V
- 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 → O(V^3 log V)
    - Bellman Ford O(V * V * E), if E = V^2 → O(V^4)
    - Slow to code
  - Use Floyd Warshall, a clever **DP** algorithm
    - O(V^3) algorithm
    - <u>Very easy</u> to code!
    - In this problem, V is <= 100, so Floyd Warshall is DOABLE!!

Floyd Warshall – Template

- $O(V^3)$ since we have three nested loops!
- Use adjacency matrix: `G[MAX_V][MAX_V];`
  - So that weight of edge(i, j) can be accessed in O(1)
- Can use 2-dimensional matrix (instead of 3-d)
  - Dimension k can be done "on the fly"

```
REP (k, 0, V - 1)
  REP (i, 0, V - 1)
    REP (j, 0, V - 1)
      G[i][j] = min(G[i][j], G[i][k] + G[k][j]);
```

  - See more explanation of this DP in the book

Floyd Warshall Variants (1)

- Floyd Warshall's is to get all pairs shortest paths
  - But we can have some other variants of usage
- Two examples:
  1. Transitive Closure (Warshall's algorithm): 334
     - We just want to see if vertex i is connected to j

```
REP (k, 0, V - 1) // O(V^3)
  REP (i, 0, V - 1)
    REP (j, 0, V - 1)
      d[i][j] = d[i][j] | (d[i][k] & d[k][j]);
```

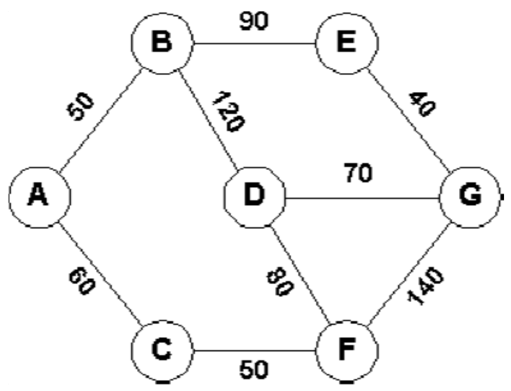
     - Logical bitwise operation is faster than arithmetic operation

Floyd Warshall Variants (2)

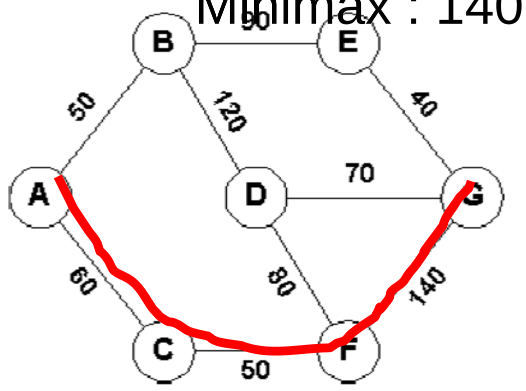
2. Minimax: 534, 10048; & its reverse Maximin: 544, 10099
   - For a path from i to j, we pick the max edge weight
   - We want to minimize max edge weight among all possible paths from i to j

```
REP (k, 0, V - 1) // O(V³)
  REP (i, 0, V - 1)
    REP (j, 0, V - 1)
      d[i][j] = min(d[i][j], max(d[i][k], d[k][j]));
```

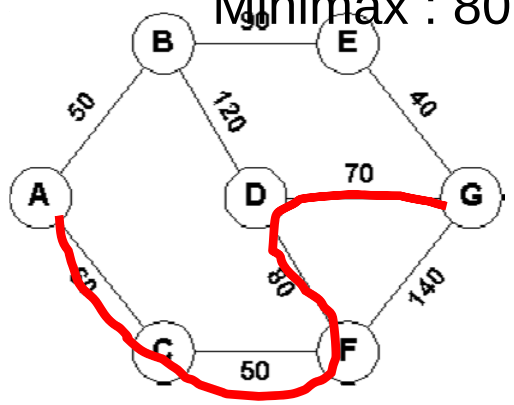
Find path from A to G that satisfy "MiniMax" criteria



Minimax : 140



Minimax : 80



Tree, Euler Graph, Directed Acyclic Graph

SPECIAL GRAPHS 1

Special Graphs in Contest

- 4 special graphs frequently appear in contest:
  - Tree, keywords: connected, E = V-1, unique path!
  - Eulerian Graph, keywords: must visit each edge once
  - Directed Acyclic Graph, keywords: no cycle
  - Bipartite, keywords: 2 sets, no edges within set!
    - » Currently not in IOI syllabus
- Some classical 'hard' problems may have *faster solution* on these special graphs
  - This allows problem setter to increase **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)…

TREE

Tree

- Tree is a special Graph. It:
  - Connected
  - Has V vertices and exactly E = V - 1 edges
  - Has no cycle
  - Has one unique path between two vertices
  - 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 (most popular one)
  - Has many practical applications:
    - Organization Tree, Directories in Operating System, etc

(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

Articulation Points & Bridges
in Tree

- In general graph
  - We need O(V+E) Tarjan's algorithm
- In tree
  - **All edges** in Tree are bridges
    - There are exactly |V|-1 of them
  - **All internal nodes** in Tree are articulation points
    - Can be easily checked by checking how many vertices with out-degree > 1

SSSP and APSP Problems
on Weighted Tree

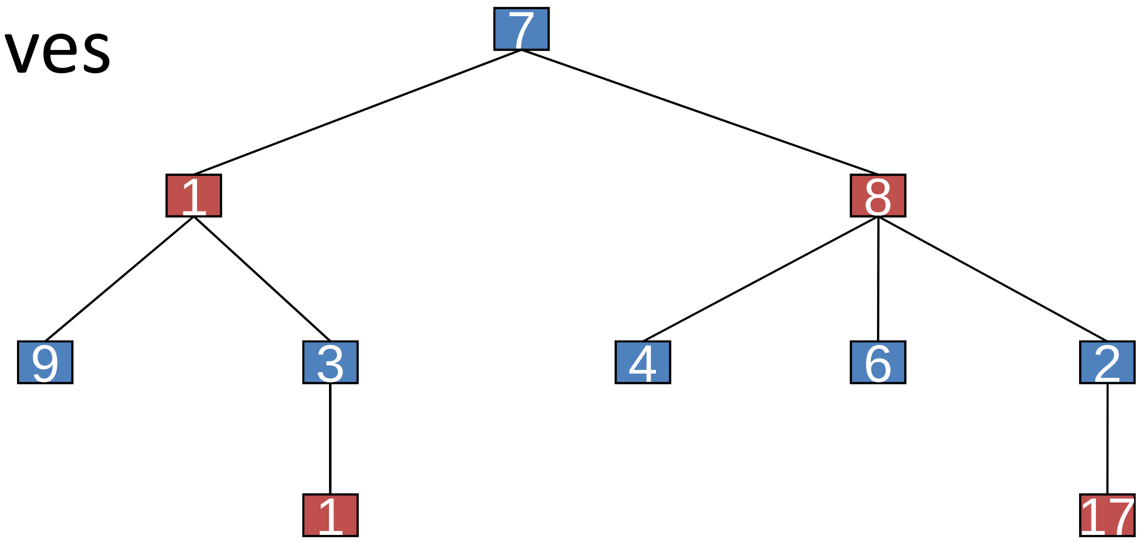
- In general weighted graph
  - SSSP problem: O((V+E) log V) Dijkstra's or O(VE) Bellman Ford's
  - APSP problem: O(V^3) Floyd Warshall's
- In weighted tree
  - SSSP problem: O(V+E = V+V = V) DFS or BFS
    - There is only 1 unique path between 2 vertices in tree
  - APSP problem: simple V calls of DFS or BFS: O(V^2)
    - But can be made even faster using LCA…

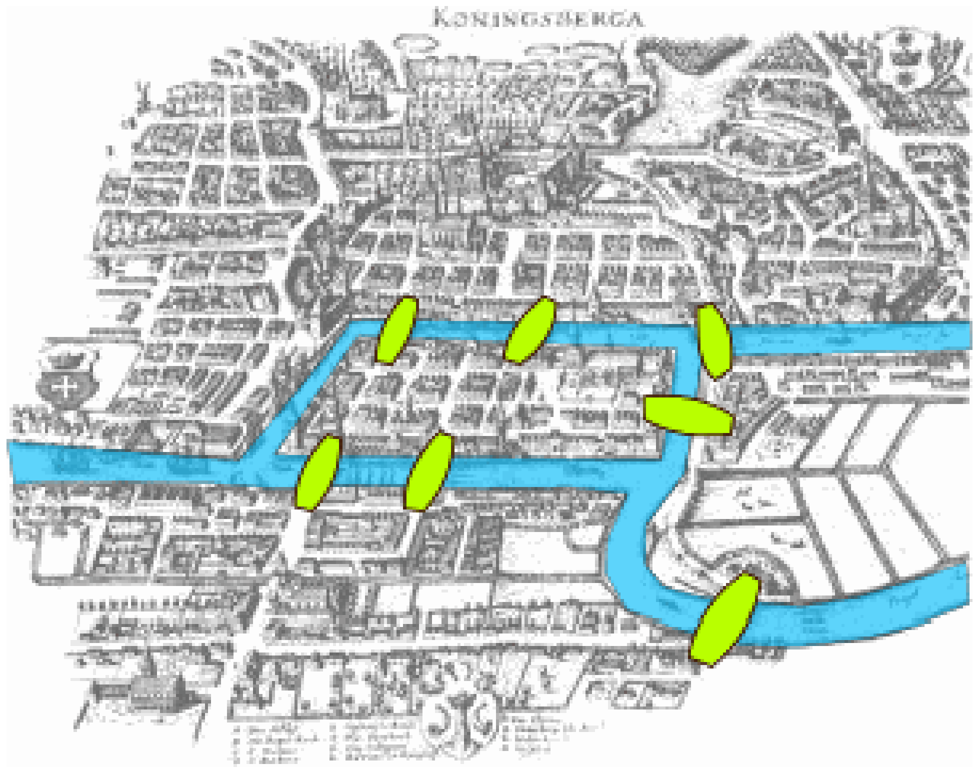
Diameter
of a Tree

- In general weighted graph
  - We have to run O(V^3) Floyd Warshall's and pick the maximum over all dist[i][j] that is not INF
- In weighted tree
  - Do DFS/BFS twice!
    - From any vertex s, find furthest vertex x with DFS/BFS
    - Then from vertex x, find furthest vertex y with DFS/BFS
    - Answer is the path length of x-y
    - O(V+E = V+V = V) only – two calls of DFS/BFS

Dynamic Programming
on Tree

- Recall the Max Weighted Independent Set problem on Tree few weeks ago
- Tree is friendly towards DP solution ☺
  - Subproblems: Subtrees
  - Base cases: Leaves





Euler Graph

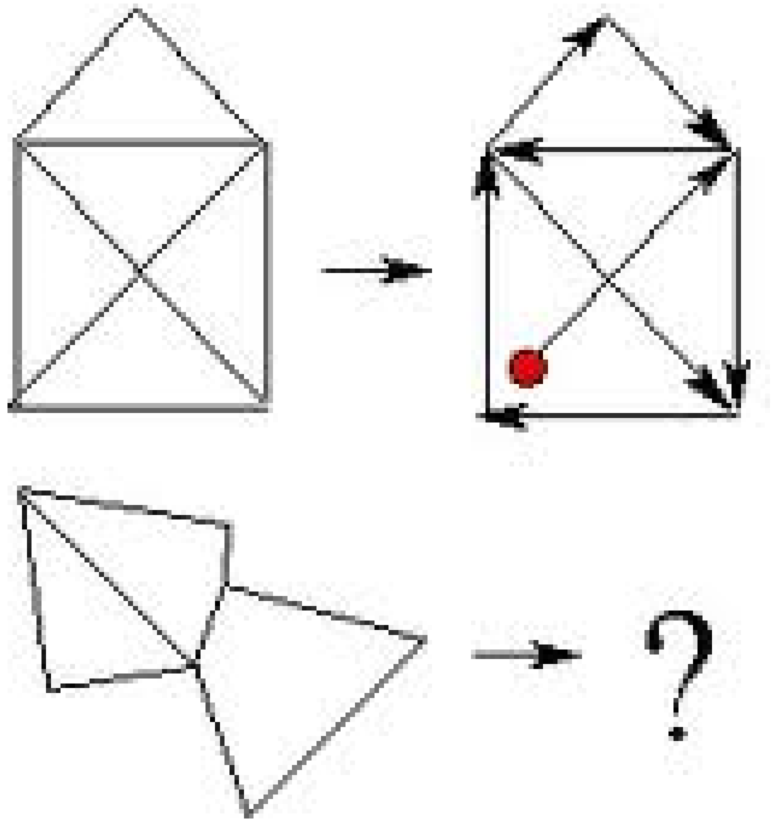
Eulerian Graph (1)

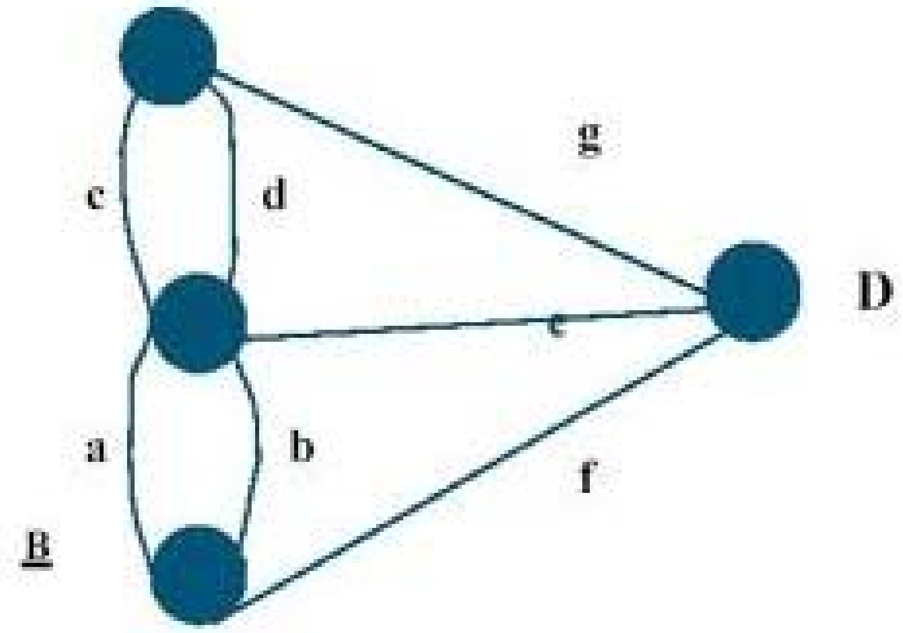
- An **Euler path** is defined as a path in a graph which visits each edge exactly once
- An **Euler tour/cycle** is an Euler path which starts and ends on the same vertex
- A graph which has either Euler path or Euler tour is called Eulerian graph

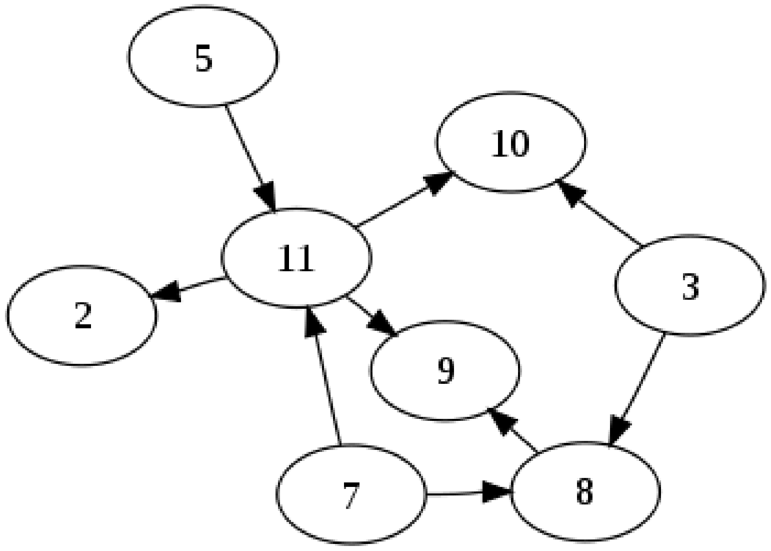
Eulerian Graph (2)

- To check whether an undirected graph has an Euler tour is **simple** ☺
  - Check if all its vertices have even degrees.
- Similarly for the Euler path
  - An undirected graph has an Euler path if all except two vertices have even degrees and at most two vertices have odd degrees. This Euler path will start from one of these odd degree vertices and end in the other
- Such degree check can be done in O(V + E), usually done simultaneously when reading the input graph

Eulerian Graph (3)







DIRECTED ACYCLIC GRAPH (DAG)

Directed Acyclic Graph (DAG)

- Some algorithms become simpler when used on DAGs instead of general graphs, based on the principle of topological ordering. For example, it is possible to find shortest paths and longest paths from a given starting vertex in DAGs in **linear time** by processing the vertices in a **topological order**, and calculating the path length for each vertex to be the minimum or maximum length obtained via any of its incoming edges. In contrast, for arbitrary graphs the shortest path may require slower algorithms such as Dijkstra's algorithm or the Bellman-Ford algorithm, and longest paths in arbitrary graphs are NP-hard to find.

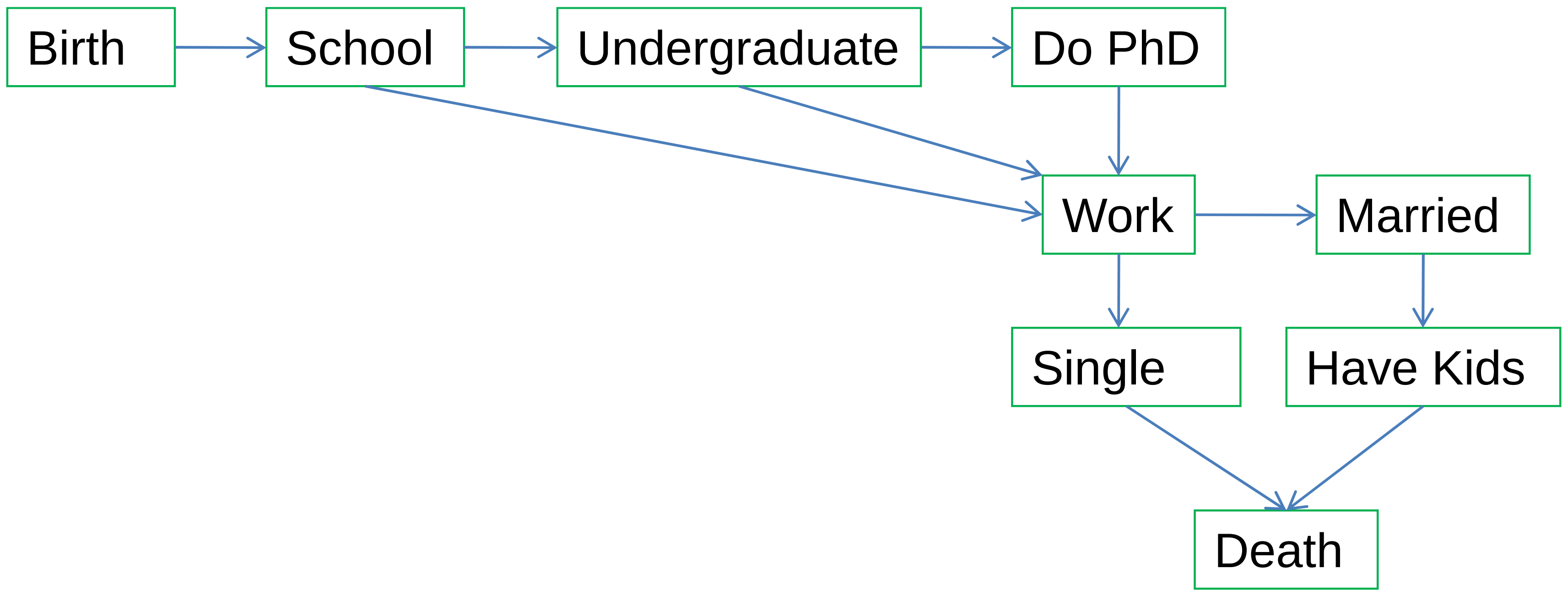
Single-Source Shortest Paths
in DAG

- In general weighted graph
  - Again 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)

Single-Source Longest Paths
in DAG

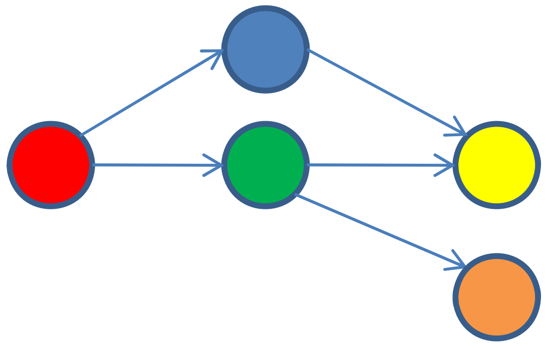
- In general weighted graph
  - This is NP complete
- In DAG
  - The solution is the same as shortest paths in DAG, just that we have tweak the relax operator
    (or alternatively, negate all edge weight in DAG)

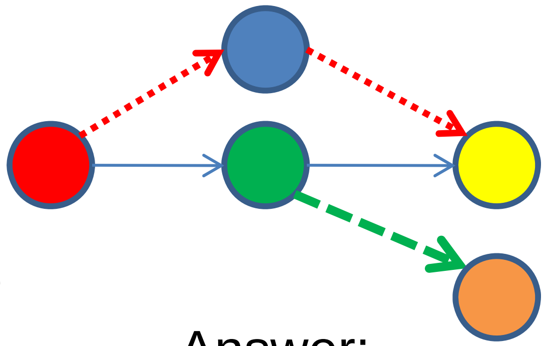
Counting Paths in DAG



Min Path Cover
in DAG

- Illustration:
  - Imagine that vertices are passengers, and draw edge between two vertices if a single taxi can satisfy the demand of both passengers on time…
  - What is the minimum number of taxis that must be deployed to serve all passengers?
- This problem is called: Min Path Cover
  - Set of directed paths s.t. every vertex in the graph belong to at least one path (including path of length 0, i.e. a single vertex)

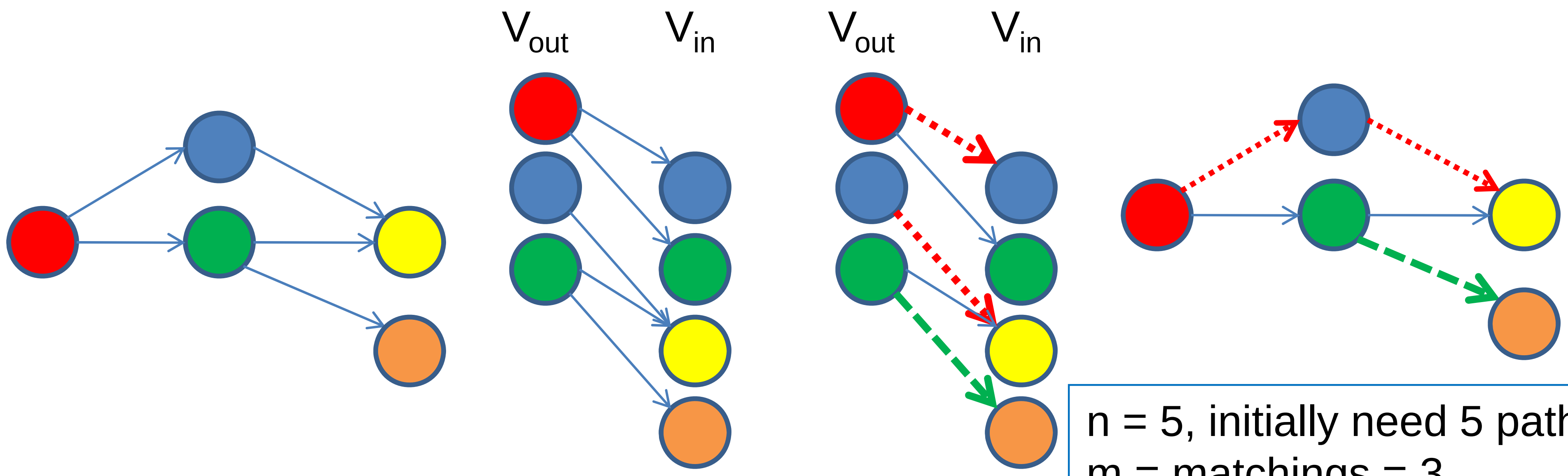




Answer:
2 Taxis!

Min Path Cover
in DAG

- This problem is NP-Complete in general graph
- In DAG, we can reduce this to bipartite matching!



n = 5, initially need 5 paths
m = matchings = 3
each matching reduce the need of one path
n – m = 5 – 3 = 2 paths can cover all vertices

BE A PROBLEM SETTER

Be a Problem Setter

- Problem Solver:
  A. Read the problem
  B. Think of a good algorithm
  C. Write 'solution'
  D. Create tricky I/O
  E. If WA, go to A/B/C/D
  F. If TLE/MLE, go to A/B/C/D
  G. If AC, stop ☺

- Problem Setter:
  A. Write a <u>good</u> problem
  B. Write a <u>good</u> solution
  C. Set a <u>good</u> secret I/O
  D. Set problem settings
- A problem setter <u>must</u> think from a different angle!
  – By setting good problems, you will simultaneously be a better problem solver!!

Problem Setter Tasks (1)

- Write a <u>good</u> problem
  - Options:
    - Pick an algorithm, then find problem/story, or
    - Find a problem/story, then identify a good algorithm for it (harder)
  - Problem description must not be <u>ambiguous</u>
    - Specify input constraints
    - Good English!
    - Easy one: longer, Hard one: shorter!
- Write a <u>good</u> solution
  - Must be able to solve your own problem!
    - To set hard problem, one must increase his own programming skill!
  - Use the <u>best possible</u> algorithm with lowest time complexity
    - Not just the one 'that works'...

Problem Setter Tasks (2)

- Set a good secret I/O
  - Tricky test cases to check AC vs WA
    - Usually ‘boundary case’
  - Large test cases to check AC vs TLE/MLE
    - Perhaps use input generator to generate large test case, then pass this large input to our correct solution

- Set problem settings
  - Time Limit:
    - Usually 2 or 3 times the timings of your own best solutions
      - Java slower than C++!
  - Memory Limit:
    - Check OJ setting^
  - Problem Name:
    - Avoid revealing the algorithm in the problem name

FYI: Be A Contest Organizer

- Contest Organizer Tasks:
  - Set problems of *various* topic
    - Better set by >1 problem setter
  - Must balance the difficulty of the problem set
    - Try to make it fun
    - Each team solves some problems
    - Each problem is solved by some teams
    - No team solve all problems
      - Every teams must work until the end of contest

Special Homework

- Create **one** problem of your choice
  - Can be of **any** problem type
    - Regardless we have studied that in CS3233 or not
- Deliverables:
  - 1 html: problem description + sample I/O
  - 1 source code: cpp/java
  - 1 test data file (of multiple instances)
  - 1 short txt write up of the expected solution
  - Zip and upload your problem into special folder in IVLE
    - CS3233 / Homework / Be A Problem Setter

Summary

- Today, we have gone through **few more** popular graph problems & algorithms
  - Dijkstra's, Bellman Ford's, Floyd Warshall's, and some special graphs
    - Note that knowing these algorithm will not be too useful in contest setting, you have to practice using them, at least code each of this algorithm once on a contest problem!