Exercise 4.2.9.1: Prove (or disprove) this statement: "If two vertices are in the same SCC, then there is no path between them that ever leaves the SCC"!

Exercise 4.2.9.2*: Write a code that takes in a Directed Graph and then convert it into a Directed Acyclic Graph (DAG) by contracting the SCCs (e.g Figure 4.9, top to bottom)! See Section 8.4.3 for a sample application.

Remarks About Graph Traversal in Programming Contests

It is remarkable that the simple DFS and BFS traversal algorithms have so many interesting variants that can be used to solve various graph problems on top of their basic form for traversing a graph. In ICPC, any of these variants can appear. In IOI, creative tasks involving graph traversal can appear.

Using DFS (or BFS) to find connected components in an undirected graph is rarely asked per se although its variant: flood fill, is one of the most frequent problem type *in the past*. However, we feel that the number of (new) flood fill problems is getting smaller.

Topological sort is rarely used per se, but it is a useful pre-processing step for 'DP on (implicit) DAG', see Section 4.7.1. The simplest version of topological sort code is very easy to memorize as it is just a simple DFS variant. The alternative Kahn's algorithm (the 'modified BFS' that only enqueue vertices with 0-incoming degrees) is also equally simple.

Efficient $O(V + E)$ solutions for bipartite graph check, graph edges property check, and finding articulation points/bridges are good to know but as seen in the UVa online judge (and recent ICPC regionals in Asia), not many problems use them now.

The knowledge of Tarjan's SCC algorithm may come in handy to solve modern problems where one of its sub-problem involves directed graphs that 'requires transformation' to DAG by contracting cycles—see Section 8.4.3. The library code shown in this book may be something that you should bring into a programming contest that allows hard copy printed library code like ICPC. However in IOI, the topic of Strongly Connected Component is currently excluded from the IOI 2009 syllabus [20].

Although many of the graph problems discussed in this section can be solved by either DFS or BFS. Personally, we feel that many of them are easier to be solved using the recursive and more memory friendly DFS. We do not normally use BFS for pure graph traversal problems but we will use it to solve the Single-Source Shortest Paths problems on unweighted graph (see Section 4.4). Table 4.2 shows important comparison between these two popular graph traversal algorithms.

	$O(V + E)$ **DFS**	$O(V + E)$ **BFS**
Pros	*Usually* use less memory	Can solve SSSP
	Can find articulation points, bridges, SCC	(on unweighted graphs)
Cons	Cannot solve SSSP	*Usually* use more memory
	on unweighted graphs	(bad for large graph)
Code	Slightly easier to code	Just a bit longer to code

Table 4.2: Graph Traversal Algorithm Decision Table

We have provided the animation of DFS/BFS algorithm and (some of) their variants in the URL below. Use it to further strengthen your understanding of these algorithms.

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/dfsbfs.html`

Programming Exercises related to Graph Traversal:

- Just Graph Traversal
  1. UVa 00118 - Mutant Flatworld Explorers (traversal on *implicit* graph)
  2. UVa 00168 - Theseus and the ... (Adjacency Matrix, parsing, traversal)
  3. UVa 00280 - Vertex (graph, reachability test by traversing the graph)
  4. *UVa 00318 - Domino Effect* (traversal, be careful of corner cases)
  5. UVa 00614 - Mapping the Route (traversal on *implicit* graph)
  6. UVa 00824 - Coast Tracker (traversal on *implicit* graph)
  7. UVa 10113 - Exchange Rates (just graph traversal, but uses fraction and gcd, see the relevant sections in Chapter 5)
  8. UVa 10116 - Robot Motion (traversal on *implicit* graph)
  9. UVa 10377 - Maze Traversal (traversal on *implicit* graph)
  10. UVa 10687 - Monitoring the Amazon (build graph, geometry, reachability)
  11. **UVa 11831 - Sticker Collector ... *** (*implicit* graph; input order is 'NSEW'!)
  12. UVa 11902 - Dominator (disable vertex one by one, check if the reachability from vertex 0 changes)
  13. **UVa 11906 - Knight in a War Grid *** (DFS/BFS for reachability, several tricky cases; be careful when M = 0 || N = 0 || M = N)
  14. *UVa 12376 - As Long as I Learn, I Live* (simulated greedy traversal on DAG)
  15. ***UVa 12442 - Forwarding Emails ****** (modified DFS, special graph)
  16. *UVa 12582 - Wedding of Sultan* (given graph DFS traversal, count the degree of each vertex)
  17. IOI 2011 - Tropical Garden (graph traversal; DFS; involving cycle)
- Flood Fill/Finding Connected Components
  1. UVa 00260 - Il Gioco dell'X (6 neighbors per cell!)
  2. UVa 00352 - The Seasonal War (count # of connected components (CC))
  3. UVa 00459 - Graph Connectivity (also solvable with 'union find')
  4. UVa 00469 - Wetlands of Florida (count size of a CC; discussed in this section)
  5. UVa 00572 - Oil Deposits (count number of CCs, similar to UVa 352)
  6. UVa 00657 - The Die is Cast (there are three 'colors' here)
  7. *UVa 00722 - Lakes* (count the size of CCs)
  8. *UVa 00758 - The Same Game* (floodfill++)
  9. UVa 00776 - Monkeys in a Regular ... (label CCs with indices, format output)
  10. UVa 00782 - Countour Painting (replace ' ' with '#' in the grid)
  11. UVa 00784 - Maze Exploration (very similar with UVa 782)
  12. UVa 00785 - Grid Colouring (also very similar with UVa 782)
  13. UVa 00852 - Deciding victory in Go (interesting board game 'Go')
  14. UVa 00871 - Counting Cells in a Blob (find the size of the largest CC)
  15. ***UVa 01103 - Ancient Messages ****** (LA 5130, World Finals Orlando11; major hint: each hieroglyph has unique number of white connected component; then it is an implementation exercise to parse the input and run flood fill to determine the number of white CC inside each black hieroglyph)
  16. UVa 10336 - Rank the Languages (count and rank CCs with similar color)

17. UVa 10707 - 2D - Nim (check graph isomorphism; a tedious problem; involving connected components)
18. UVa 10946 - You want what filled? (find CCs and rank them by their size)
19. **UVa 11094 - Continents *** (tricky flood fill as it involves scrolling)
20. UVa 11110 - Equidivisions (flood fill + satisfy the constraints given)
21. UVa 11244 - Counting Stars (count number of CCs)
22. UVa 11470 - Square Sums (you can do 'flood fill' layer by layer; however, there is other way to solve this problem, e.g. by finding the patterns)
23. UVa 11518 - Dominos 2 (unlike UVa 11504, we treat SCCs as simple CCs)
24. UVa 11561 - Getting Gold (flood fill with extra blocking constraint)
25. UVa 11749 - Poor Trade Advisor (find largest CC with highest average PPA)
26. **UVa 11953 - Battleships *** (interesting twist of flood fill problem)

- Topological Sort

1. UVa 00124 - Following Orders (use backtracking to generate valid toposorts)
2. UVa 00200 - Rare Order (toposort)
3. **UVa 00872 - Ordering *** (similar to UVa 124, use backtracking)
4. **UVa 10305 - Ordering Tasks *** (run toposort algorithm in this section)
5. **UVa 11060 - Beverages *** (must use Kahn's algorithm—the 'modified BFS' topological sort)
6. UVa 11686 - Pick up sticks (toposort + cycle check)
   Also see: DP on (implicit) DAG problems (see Section 4.7.1)

- Bipartite Graph Check

1. **UVa 10004 - Bicoloring *** (bipartite graph check)
2. UVa 10505 - Montesco vs Capuleto (bipartite graph, take max(left, right))
3. **UVa 11080 - Place the Guards *** (bipartite graph check, some tricky cases)
4. **UVa 11396 - Claw Decomposition *** (it is just a bipartite graph check)

- Finding Articulation Points/Bridges

1. **UVa 00315 - Network *** (finding articulation points)
2. UVa 00610 - Street Directions (finding bridges)
3. **UVa 00796 - Critical Links *** (finding bridges)
4. UVa 10199 - Tourist Guide (finding articulation points)
5. ***UVa 10765 - Doves and Bombs **** (finding articulation points)

- Finding Strongly Connected Components

1. **UVa 00247 - Calling Circles *** (SCC + printing solution)
2. UVa 01229 - Sub-dictionary (LA 4099, Iran07, identify the SCC of the graph; these vertices and the vertices that have path towards them (e.g. needed to understand these words too) are the answers of the question)
3. UVa 10731 - Test (SCC + printing solution)
4. **UVa 11504 - Dominos *** (interesting problem: count |SCCs| without incoming edge from a vertex outside that SCC)
5. UVa 11709 - Trust Groups (find number of SCC)
6. UVa 11770 - Lighting Away (similar to UVa 11504)
7. **UVa 11838 - Come and Go *** (check if graph is strongly connected)

4.3 Minimum Spanning Tree

4.3.1 Overview and Motivation

Motivating problem: Given a connected, undirected, and weighted graph G (see the leftmost graph in Figure 4.10), select a subset of edges $E' \in G$ such that the graph G is (still) connected and the total weight of the selected edges E' is minimal!

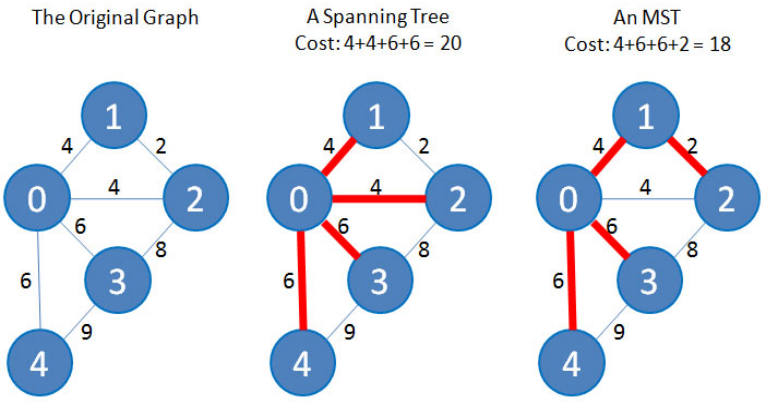


Figure 4.10: Example of an MST Problem

To satisfy the connectivity criteria, we need at least $V-1$ edges that form a *tree* and this tree must spans (covers) all $V \in G$—the *spanning tree*! There can be several valid spanning trees in G, i.e. see Figure 4.10, middle and right sides. The DFS and BFS spanning trees that we have learned in previous Section 4.2 are also possible. Among these possible spanning trees, there are some (at least one) that satisfy the minimal weight criteria.

This problem is called the Minimum Spanning Tree (MST) problem and has many practical applications. For example, we can model a problem of building road network in remote villages as an MST problem. The vertices are the villages. The edges are the potential roads that may be built between those villages. The cost of building a road that connects village i and j is the weight of edge (i, j). The MST of this graph is therefore the minimum cost road network that connects all these villages. In UVa online judge [47], we have some basic MST problems like this, e.g. UVa 908, 1174, 1208, 10034, 11631, etc.

This MST problem can be solved with several well-known algorithms, i.e. Prim's and Kruskal's. Both are Greedy algorithms and explained in many CS textbooks [7, 58, 40, 60, 42, 1, 38, 8]. The MST weight produced by these two algorithms is unique, but there can be more than one spanning tree that have the same MST weight.

4.3.2 Kruskal's Algorithm

Joseph Bernard *Kruskal* Jr.'s algorithm first sorts E edges based on non decreasing weight. This can be easily done by storing the edges in an EdgeList data structure (see Section 2.4.1) and then sort the edges based on non-decreasing weight. Then, Kruskal's algorithm *greedily* tries to add each edge into the MST as long as such addition does not form a cycle. This cycle check can be done easily using the lightweight Union-Find Disjoint Sets discussed in Section 2.4.2. The code is short (because we have separated the Union-Find Disjoint Sets implementation code in a separate class). The overall runtime of this algorithm is $O($sorting + trying to add each edge $\times$ cost of Union-Find operations$) = O(E \log E + E \times (\approx 1)) = O(E \log E) = O(E \log V^2) = O(2 \times E \log V) = O(E \log V)$.

```
// inside int main()
  vector< pair<int, ii> > EdgeList;   // (weight, two vertices) of the edge
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);            // read the triple: (u, v, w)
    EdgeList.push_back(make_pair(w, ii(u, v))); }             // (w, u, v)
  sort(EdgeList.begin(), EdgeList.end()); // sort by edge weight O(E log E)
                      // note: pair object has built-in comparison function
  int mst_cost = 0;
  UnionFind UF(V);                        // all V are disjoint sets initially
  for (int i = 0; i < E; i++) {                       // for each edge, O(E)
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.second)) {  // check
      mst_cost += front.first;                // add the weight of e to MST
      UF.unionSet(front.second.first, front.second.second);    // link them
  } }                         // note: the runtime cost of UFDS is very light
  // note: the number of disjoint sets must eventually be 1 for a valid MST
  printf("MST cost = %d (Kruskal's)\n", mst_cost);
```

Figure 4.11 shows step by step execution of Kruskal's algorithm on the graph shown in Figure 4.10—leftmost. Notice that the final MST is not unique.

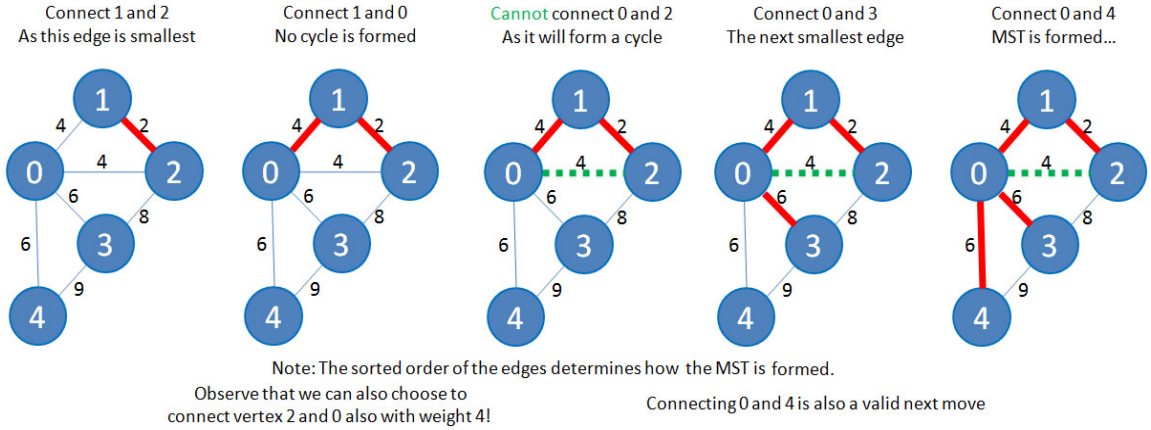


Figure 4.11: Animation of Kruskal's Algorithm for an MST Problem

Exercise 4.3.2.1: The code above only stops after the last edge in EdgeList is processed. In many cases, we can stop Kruskal's *earlier*. Modify the code to implement this!

Exercise 4.3.2.2*: Can you solve the MST problem *faster* than $O(E \log V)$ if the input graph is guaranteed to have edge weights that lie between a small integer range of `[0..100]`? Is the potential speed-up significant?

4.3.3 Prim's Algorithm

Robert Clay *Prim*'s algorithm first takes a starting vertex (for simplicity, we take vertex 0), flags it as 'taken', and enqueues a pair of information into a priority queue: The weight w and the other end point u of the edge $0 \rightarrow u$ that is not taken yet. These pairs are sorted in the priority queue based on increasing weight, and if tie, by increasing vertex number. Then, Prim's algorithm *greedily* selects the pair (w, u) in front of the priority

queue—which has the minimum weight w—if the end point of this edge—which is u—has not been taken before. This is to prevent cycle. If this pair (w, u) is valid, then the weight w is added into the MST cost, u is marked as taken, and pair (w', v) of each edge $u \rightarrow v$ with weight w' that is incident to u is enqueued into the priority queue if v has not been taken before. This process is repeated until the priority queue is empty. The code length is about the same as Kruskal's and also runs in O(process each edge once $\times$ cost of enqueue/dequeue) $= O(E \times \log E) = O(E \log V)$.

```
vi taken;                                  // global boolean flag to avoid cycle
priority_queue<ii> pq;       // priority queue to help choose shorter edges
          // note: default setting for C++ STL priority_queue is a max heap
void process(int vtx) {    // so, we use -ve sign to reverse the sort order
  taken[vtx] = 1;
  for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
    ii v = AdjList[vtx][j];
    if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
} }                                  // sort by (inc) weight then by (inc) id

// inside int main()---assume the graph is stored in AdjList, pq is empty
  taken.assign(V, 0);                  // no vertex is taken at the beginning
  process(0);   // take vertex 0 and process all edges incident to vertex 0
  mst_cost = 0;
  while (!pq.empty()) {  // repeat until V vertices (E=V-1 edges) are taken
    ii front = pq.top(); pq.pop();
    u = -front.second, w = -front.first;  // negate the id and weight again
    if (!taken[u])                   // we have not connected this vertex yet
      mst_cost += w, process(u); // take u, process all edges incident to u
  }                                            // each edge is in pq only once!
  printf("MST cost = %d (Prim's)\n", mst_cost);
```

Figure 4.12 shows the step by step execution of Prim's algorithm on the same graph shown in Figure 4.10—leftmost. Please compare it with Figure 4.11 to study the similarities and differences between Kruskal's and Prim's algorithms.

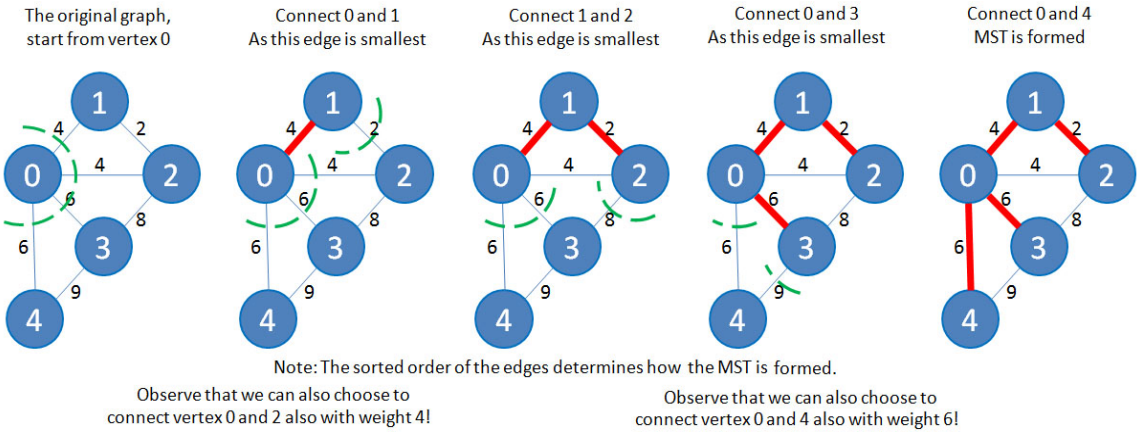


Figure 4.12: Animation of Prim's Algorithm for the same graph as in Figure 4.10—left

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/mst.html`

Source code: `ch4_03_kruskal_prim.cpp/java`

4.3.4 Other Applications

Variants of basic MST problem are interesting. In this section, we will explore some of them.

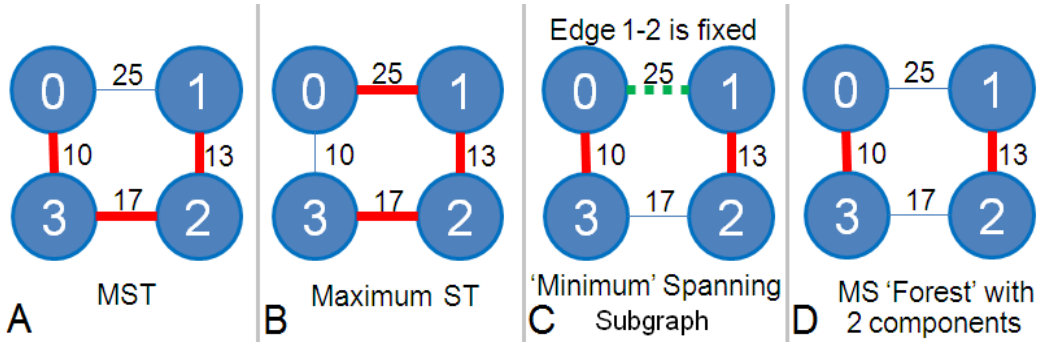


Figure 4.13: From left to right: MST, 'Maximum' ST, 'Minimum' SS, MS 'Forest'

'Maximum' Spanning Tree

This is a simple variant where we want the maximum instead of the minimum ST, for example: UVa 1234 - RACING (note that this problem is written in such a way that it does not look like an MST problem). In Figure 4.13.B, we see an example of a Maximum ST. Compare it with the corresponding MST (Figure 4.13.A).

The solution for this variant is very simple: Modify Kruskal's algorithm a bit, we now simply sort the edges based on *non increasing* weight.

'Minimum' Spanning Subgraph

In this variant, we do not start with a clean slate. Some edges in the given graph have already been fixed and must be taken as part of the solution, for example: UVa 10147 - Highways. These default edges may form a non-tree in the first place. Our task is to continue selecting the remaining edges (if necessary) to make the graph connected in the least cost way. The resulting Spanning Subgraph may not be a tree and even if it is a tree, it may not be the MST. That's why we put the term 'Minimum' in quotes and use the term 'subgraph' rather than 'tree'. In Figure 4.13.C, we see an example when one edge 0-1 is already fixed. The actual MST is 10+13+17 = 40 which omits the edge 0-1 (Figure 4.13.A). However, the solution for this example must be (25)+10+13 = 48 which uses the edge 0-1.

The solution for this variant is simple. After taking into account all the fixed edges and their costs, we continue running Kruskal's algorithm on the remaining free edges until we have a spanning subgraph (or spanning tree).

Minimum 'Spanning Forest'

In this variant, we want to form a forest of K connected components (K subtrees) in the least cost way where K is given beforehand in the problem description, for example: UVa 10369 - Arctic Networks. In Figure 4.13.A, we observe that the MST for this graph is 10+13+17 = 40. But if we are happy with a spanning forest with 2 connected components, then the solution is just 10+13 = 23 on Figure 4.13.D. That is, we omit the edge 2-3 with weight 17 which will connect these two components into one spanning tree if taken.

To get the minimum spanning forest is simple. Run Kruskal's algorithm as per normal, but as soon as the number of connected components equals to the desired pre-determined number K, we can terminate the algorithm.

Second Best Spanning Tree

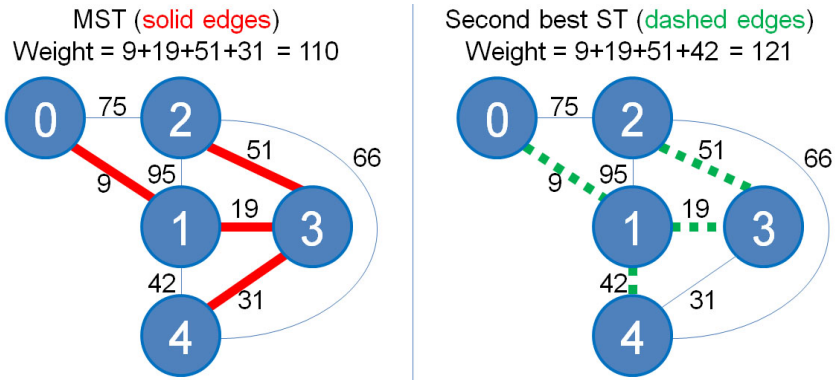


Figure 4.14: Second Best ST (from UVa 10600 [47])

Sometimes, alternative solutions are important. In the context of finding the MST, we may want not just the MST, but also the second best spanning tree, in case the MST is not workable, for example: UVa 10600 - ACM contest and blackout. Figure 4.14 shows the MST (left) and the second best ST (right). We can see that the second best ST is actually the MST with just two edges difference, i.e. one edge is taken out from the MST and another chord[5] edge is added into the MST. Here, edge 3-4 is taken out and edge 1-4 is added in.

A solution for this variant is a modified Kruskal's: Sort the edges in $O(E \log E) = O(E \log V)$, then find the MST using Kruskal's in $O(E)$. Next, for each edge in the MST (there are at most V-1 edges in the MST), temporarily flag it so that it cannot be chosen, then try to find the MST again in $O(E)$ but now *excluding* that flagged edge. Note that we do not have to re-sort the edges at this point. The best spanning tree found after this process is the second best ST. Figure 4.15 shows this algorithm on the given graph. In overall, this algorithm runs in $O(\text{sort the edges once} + \text{find the original MST} + \text{find the second best ST}) = O(E \log V + E + VE) = O(VE)$.

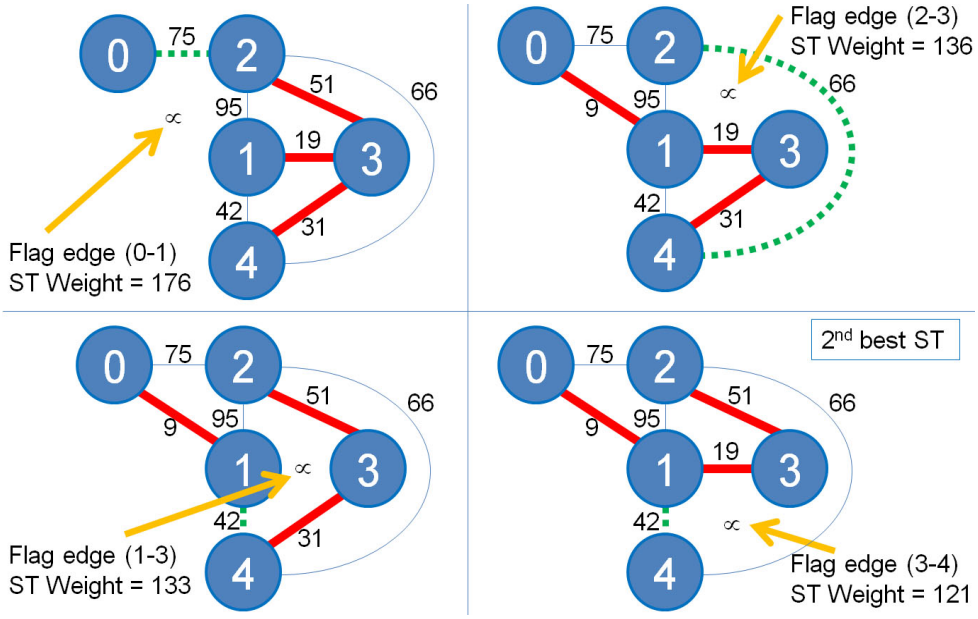


Figure 4.15: Finding the Second Best Spanning Tree from the MST

[5]A chord edge is defined as an edge in graph G that is not selected in the MST of G.

Minimax (and Maximin)

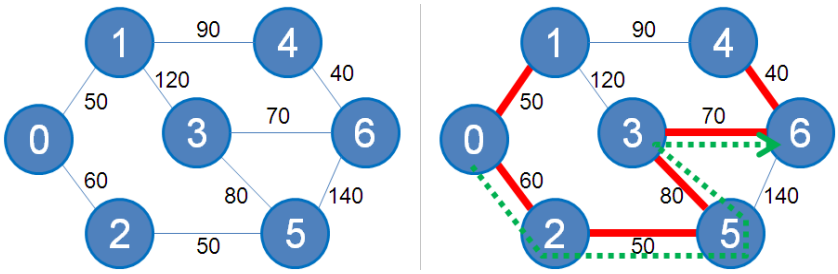


Figure 4.16: Minimax (UVa 10048 [47])

The minimax path problem is a problem of finding the minimum of maximum edge weight among all possible paths between two vertices i to j. The cost for a path from i to j is determined by the maximum edge weight along this path. Among all these possible paths from i to j, pick the one with the minimum max-edge-weight. The reverse problem of maximin is defined similarly.

The minimax path problem between vertex i and j can be solved by modeling it as an MST problem. With a rationale that the problem prefers a path with low individual edge weights even if the path is longer in terms of number of vertices/edges involved, then having the MST (using Kruskal's or Prim's) of the given weighted graph is a correct step. The MST is connected thus ensuring a path between any pair of vertices. The minimax path solution is thus the max edge weight along the unique path between vertex i and j in this MST.

The overall time complexity is O(build MST + one traversal on the resulting tree). As $E = V - 1$ in a tree, any traversal on tree is just $O(V)$. Thus the complexity of this approach is $O(E \log V + V) = O(E \log V)$.

Figure 4.16—left is a sample test case of UVa 10048 - Audiophobia. We have a graph with 7 vertices and 9 edges. The 6 chosen edges of the MST are shown as thick lines in Figure 4.16, right. Now, if we are asked to find the minimax path between vertex 0 and 6 in Figure 4.16, right, we simply traverse the MST from vertex 0 to 6. There will only be one way, path: 0-2-5-3-6. The maximum edge weight found along the path is the required minimax cost: 80 (due to edge 5-3).

Exercise 4.3.4.1: Solve the five MST problem variants above using Prim's algorithm instead. Which variant(s) is/are not Prim's-friendly?

Exercise 4.3.4.2*: There are better solutions for the Second Best ST problem shown above. Solve this problem with a solution that is better than $O(VE)$. Hints: You can use either Lowest Common Ancestor (LCA) or Union-Find Disjoint-Sets.

Remarks About MST in Programming Contests

To solve many MST problems in today's programming contests, we can rely on Kruskal's algorithm alone and skip Prim's (or other MST) algorithm. Kruskal's is by our reckoning the best algorithm to solve programming contest problems involving MST. It is easy to understand and links well with the Union-Find Disjoint Sets data structure (see Section 2.4.2) that is used to check for cycles. However, as we do love choices, we also include the discussion of the other popular algorithm for MST: Prim's algorithm.

The default (and the most common) usage of Kruskal's (or Prim's) algorithm is to solve the Minimum ST problem (UVa 908, 1174, 1208, 11631), but the easy variant of 'Maximum' ST is also possible (UVa 1234, 10842). Note that most (if not all) MST problems

in programming contests only ask for the *unique* MST cost and not the actual MST itself. This is because there can be different MSTs with the same minimum cost—usually it is too troublesome to write a special checker program to judge such non unique outputs.

The other MST variants discussed in this book like the 'Minimum' Spanning Subgraph (UVa 10147, 10397), Minimum 'Spanning Forest' (UVa 1216, 10369), Second best ST (UVa 10462, 10600), Minimax/Maximin (UVa 534, 544, 10048, 10099) are actually rare.

Nowadays, the more general trend for MST problems is for the problem authors to write the MST problem in such a way that it is not clear that the problem is actually an MST problem (e.g. UVa 1216, 1234, 1235). However, once the contestants spot this, the problem may become 'easy'.

Note that there are harder MST problems that may require more sophisticated algorithm to solve, e.g. Arborescence problem, Steiner tree, degree constrained MST, k-MST, etc.

Programming Exercises related to Minimum Spanning Tree:

- Standard
  1. UVa 00908 - Re-connecting ... (basic MST problem)
  2. *UVa 01174 - IP-TV* (LA 3988, SouthWesternEurope07, MST, classic, just need a mapper to map city names to indices)
  3. UVa 01208 - Oreon (LA 3171, Manila06, MST)
  4. UVa 01235 - Anti Brute Force Lock (LA 4138, Jakarta08, the underlying problem is MST)
  5. UVa 10034 - Freckles (straightforward MST problem)
  6. **UVa 11228 - Transportation System *** (split the output for short versus long edges)
  7. **UVa 11631 - Dark Roads *** (weight of (all graph edges - all MST edges))
  8. UVa 11710 - Expensive Subway (output 'Impossible' if the graph is still disconnected after running MST)
  9. UVa 11733 - Airports (maintain cost at every update)
  10. **UVa 11747 - Heavy Cycle Edges *** (sum the edge weights of the chords)
  11. UVa 11857 - Driving Range (find weight of the last edge added to MST)
  12. IOI 2003 - Trail Maintenance (use efficient incremental MST)
- Variants
  1. UVa 00534 - Frogger (minimax, also solvable with Floyd Warshall's)
  2. UVa 00544 - Heavy Cargo (maximin, also solvable with Floyd Warshall's)
  3. *UVa 01160 - X-Plosives* (count the number of edges not taken by Kruskal's)
  4. UVa 01216 - The Bug Sensor Problem (LA 3678, Kaohsiung06, minimum 'spanning forest')
  5. UVa 01234 - RACING (LA 4110, Singapore07, 'maximum' spanning tree)
  6. **UVa 10048 - Audiophobia *** (minimax, see the discussion above)
  7. UVa 10099 - Tourist Guide (maximin, also solvable with Floyd Warshall's)
  8. UVa 10147 - Highways ('minimum' spanning subgraph)
  9. **UVa 10369 - Arctic Networks *** (minimum spanning 'forest')
  10. UVa 10397 - Connect the Campus ('minimum' spanning subgraph)
  11. UVa 10462 - Is There A Second ... (second best spanning tree)
  12. **UVa 10600 - ACM Contest and ... *** (second best spanning tree)
  13. UVa 10842 - Traffic Flow (find min weighted edge in 'max' spanning tree)

Profile of Algorithm Inventors

Robert Endre Tarjan (born 1948) is an American computer scientist. He is the discoverer of several important graph algorithms. The most important one in the context of competitive programming is the algorithm for finding **Strongly Connected Components algorithm** in a directed graph and the algorithm to find **Articulation Points and Bridges** in an undirected graph (discussed in Section 4.2 together with other DFS variants invented by him and his colleagues [63]). He also invented **Tarjan's off-line Least Common Ancestor algorithm**, invented **Splay Tree data structure**, and analyze the time complexity of the **Union-Find Disjoint Sets data structure** (see Section 2.4.2).

John Edward Hopcroft (born 1939) is an American computer scientist. He is the Professor of Computer Science at Cornell University. Hopcroft received the Turing Award—the most prestigious award in the field and often recognized as the 'Nobel Prize of computing' (jointly with Robert Endre Tarjan in 1986)—for fundamental achievements in the design and analysis of algorithms and data structures. Along with his work with Tarjan on planar graphs (and some other graph algorithms like **finding articulation points/bridges using DFS**) he is also known for the **Hopcroft-Karp's algorithm** for finding matchings in bipartite graphs, invented together with Richard Manning Karp [28] (see Section 9.12).

Joseph Bernard Kruskal, Jr. (1928-2010) was an American computer scientist. His best known work related to competitive programming is the **Kruskal's algorithm** for computing the Minimum Spanning Tree (MST) of a weighted graph. MST have interesting applications in construction and *pricing* of communication networks.

Robert Clay Prim (born 1921) is an American mathematician and computer scientist. In 1957, at Bell Laboratories, he developed Prim's algorithm for solving the MST problem. Prim knows Kruskal as they worked together in Bell Laboratories. Prim's algorithm, was originally discovered earlier in 1930 by Vojtêch Jarník and rediscovered independently by Prim. Thus Prim's algorithm sometimes also known as Jarník-Prim's algorithm.

Vojtêch Jarník (1897-1970) was a Czech mathematician. He developed the graph algorithm now known as Prim's algorithm. In the era of fast and widespread publication of scientific results nowadays. Prim's algorithm would have been credited to Jarník instead of Prim.

Edsger Wybe Dijkstra (1930-2002) was a Dutch computer scientist. One of his famous contributions to computer science is the shortest path-algorithm known as **Dijkstra's algorithm** [10]. He does not like 'GOTO' statement and influenced the widespread deprecation of 'GOTO' and its replacement: structured control constructs. One of his famous Computing phrase: "two or more, use a for".

Richard Ernest Bellman (1920-1984) was an American applied mathematician. Other than inventing the **Bellman Ford's algorithm** for finding shortest paths in graphs that have negative weighted edges (and possibly negative weight cycle), Richard Bellman is more well known by his invention of the *Dynamic Programming* technique in 1953.

Lester Randolph Ford, Jr. (born 1927) is an American mathematician specializing in network flow problems. Ford's 1956 paper with Fulkerson on the maximum flow problem and the **Ford Fulkerson's method** for solving it, established the max-flow min-cut theorem.

Delbert Ray Fulkerson (1924-1976) was a mathematician who co-developed the **Ford Fulkerson's method**, an algorithm to solve the Max Flow problem in networks. In 1956, he published his paper on the Ford Fulkerson's method together with Lester R. Ford.

4.4 Single-Source Shortest Paths

4.4.1 Overview and Motivation

Motivating problem: Given a *weighted* graph G and a starting source vertex s, what are the *shortest paths* from s to *every other vertices* of G?

This problem is called the *Single-Source*[6] *Shortest Paths* (SSSP) problem on a *weighted graph*. It is a classical problem in graph theory and has many real life applications. For example, we can model the city that we live in as a graph. The vertices are the road junctions. The edges are the roads. The time taken to traverse a road is the weight of the edge. You are currently in one road junction. What is the shortest possible time to reach another certain road junction?

There are efficient algorithms to solve this SSSP problem. If the graph is unweighted (or all edges have equal or constant weight), we can use the efficient $O(V + E)$ BFS algorithm shown earlier in Section 4.2.2. For a general weighted graph, BFS does not work correctly and we should use algorithms like the $O((V + E) \log V)$ Dijkstra's algorithm or the $O(VE)$ Bellman Ford's algorithm. These various algorithms are discussed below.

Exercise 4.4.1.1*: Prove that the shortest path between two vertices i and j in a graph G that has no negative weight cycle must be a *simple* path (acyclic)!

Exercise 4.4.1.2*: Prove: Subpaths of shortest paths from u to v are shortest paths!

4.4.2 SSSP on Unweighted Graph

Let's revisit Section 4.2.2. The fact that BFS visits vertices of a graph layer by layer from a source vertex (see Figure 4.3) turns BFS into a natural choice to solve the SSSP problems on *unweighted* graphs. In an unweighted graph, the distance between two neighboring vertices connected with an edge is simply one unit. Therefore, the layer count of a vertex that we have seen in Section 4.2.2 is precisely the shortest path length from the source to that vertex. For example in Figure 4.3, the shortest path from vertex 5 to vertex 7, is 4, as 7 is in the fourth layer in BFS sequence of visitation starting from vertex 5.

Some programming problems require us to *reconstruct* the actual shortest path, not just the shortest path length. For example, in Figure 4.3, the shortest path from 5 to 7 is $5 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 7$. Such reconstruction is easy if we store the shortest path (actually BFS) spanning tree[7]. This can be easily done using vector of integers `vi p`. Each vertex v remembers its parent u (`p[v] = u`) in the shortest path spanning tree. For this example, vertex 7 remembers 3 as its parent, vertex 3 remembers 2, vertex 2 remembers 1, vertex 1 remembers 5 (the source). To reconstruct the actual shortest path, we can do a simple recursion from the last vertex 7 until we hit the source vertex 5. The modified BFS code (check the comments) is relatively simple:

```
void printPath(int u) {                  // extract information from 'vi p'
  if (u == s) { printf("%d", s); return; }    // base case, at the source s
  printPath(p[u]);   // recursive: to make the output format: s -> ... -> t
  printf(" %d", u); }
```

[6]This generic SSSP problem can also be used to solve the: 1). Single-Pair (or Single-Source Single-Destination) SP problem where both source + destination vertices are given and 2). Single-Destination SP problem where we just reverse the role of source/destination vertices.

[7]Reconstructing the shortest path is not shown in the next two subsections (Dijkstra's/Bellman Ford's) but the idea is the same as the one shown here (and with reconstructing DP solution in Section 3.5.1).

```
// inside int main()
  vi dist(V, INF); dist[s] = 0;         // distance from source s to s is 0
  queue<int> q; q.push(s);
  vi p;                           // addition: the predecessor/parent vector
  while (!q.empty()) {
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] == INF) {
        dist[v.first] = dist[u] + 1;
        p[v.first] = u;      // addition: the parent of vertex v.first is u
        q.push(v.first);
  } } }
  printPath(t), printf("\n");     // addition: call printPath from vertex t
```

Source code: ch4_04_bfs.cpp/java

We would like to remark that recent programming contest problems involving BFS are no longer written as straightforward SSSP problems but written in a much more creative fashion. Possible variants include: BFS on implicit graph (2D grid: UVa 10653 or 3-D grid: UVa 532), BFS with the printing of the actual shortest path (UVa 11049), BFS on graph with some blocked vertices (UVa 10977), BFS from multi-sources (UVa 11101, 11624), BFS with single destination—solved by reversing the role of source and destination (UVa 11513), BFS with non-trivial states (UVa 10150)—more such problems in Section 8.2.3, etc. Since there are many interesting variants of BFS, we recommend that the readers try to solve as many problems as possible from the programming exercises listed in this section.

Exercise 4.4.2.1: We can run BFS from > 1 sources. We call this variant the Multi-Sources Shortest Paths (MSSP) on unweighted graph problem. Try solving UVa 11101 and 11624 to get the idea of MSSP on unweighted graph. A naïve solution is to call BFS multiple times. If there are k possible sources, such solution will run in $O(k \times (V + E))$. Can you do better?

Exercise 4.4.2.2: Suggest a simple improvement to the given BFS code above if you are asked to solve the Single-Source *Single-Destination* Shortest Path problem on an unweighted graph. That's it, you are given both the source *and* the destination vertex.

Exercise 4.4.2.3: Explain the reason why we can use BFS to solve an SSSP problem on a weighted graph where *all edges* has the same weight C?

Exercise 4.4.2.4*: Given an $R \times C$ grid map like shown below, determine the shortest path from any cell labeled as 'A' to any cell labeled as 'B'. You can only walk through cells labeled with '.' in NESW direction (counted as *one* unit) and cells labeled with alphabet 'A'-'Z' (counted as *zero* unit)! Can you solve this in $O(R \times C)$?

```
.....................CCCC.      // The answer for this test case is 13 units
AAAAA................CCCC.                        // Solution: Walk east from
AAAAA.AAA............CCCC.      // the rightmost A to leftmost C in this row
AAAAAAAAA....###....CCCC.   // then walk south from rightmost C in this row
AAAAAAAAA.................                                          // down
AAAAAAAAA.................                                            // to
.......DD..............BB                      // the leftmost B in this row
```

4.4.3 SSSP on Weighted Graph

If the given graph is *weighted*, BFS does not work. This is because there can be 'longer' path(s) (in terms of number of vertices and edges involved in the path) but has smaller total weight than the 'shorter' path found by BFS. For example, in Figure 4.17, the shortest path from source vertex 2 to vertex 3 is not via direct edge $2 \rightarrow 3$ with weight 7 that is normally found by BFS, but a 'detour' path: $2 \rightarrow 1 \rightarrow 3$ with smaller total weight $2 + 3 = 5$.

To solve the SSSP problem on weighted graph, we use a *greedy* Edsger Wybe *Dijkstra*'s algorithm. There are several ways to implement this classic algorithm. In fact, Dijkstra's original paper that describes this algorithm [10] does not describe a specific implementation. Many other Computer Scientists proposed implementation variants based on Dijkstra's original work. Here we adopt one of the easiest implementation variant that uses *built-in* C++ STL `priority_queue` (or Java `PriorityQueue`). This is to keep the length of code *minimal*—a necessary feature in competitive programming.

This Dijkstra's variant maintains a **priority** queue called `pq` that stores pairs of vertex information. The first and the second item of the pair is the distance of the vertex from the source and the vertex number, respectively. This `pq` is sorted based on *increasing distance* from the source, and if tie, by vertex number. This is different from another Dijkstra's implementation that uses binary heap feature that is not supported in built-in library[8].

This `pq` only contains one item initially: The base case $(0, s)$ which is true for the source vertex. Then, this Dijkstra's implementation variant repeats the following process until `pq` is empty: It greedily takes out vertex information pair (d, u) from the front of `pq`. If the distance to u from source recorded in d greater than `dist[u]`, it ignores u; otherwise, it process u. The reason for this special check is shown below.

When this algorithm process u, it tries to relax[9] all neighbors v of u. Every time it relaxes an edge $u \rightarrow v$, it will *enqueue* a pair (newer/shorter distance to v from source, v) into `pq` and *leave the inferior pair* (older/longer distance to v from source, v) inside `pq`. This is called 'Lazy Deletion' and it causes *more than one copy* of the same vertex in `pq` with *different distances* from source. That is why we have the check earlier to process only the *first dequeued* vertex information pair which has the correct/shorter distance (other copies will have the outdated/longer distance). The code is shown below and it looks very similar to BFS and Prim's code shown in Section 4.2.2 and 4.3.3, respectively.

```
vi dist(V, INF); dist[s] = 0;                // INF = 1B to avoid overflow
priority_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));
while (!pq.empty()) {                                          // main loop
  ii front = pq.top(); pq.pop(); // greedy: get shortest unvisited vertex
  int d = front.first, u = front.second;
  if (d > dist[u]) continue;            // this is a very important check
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                    // all outgoing edges from u
    if (dist[u] + v.second < dist[v.first]) {
      dist[v.first] = dist[u] + v.second;              // relax operation
      pq.push(ii(dist[v.first], v.first));
} } }     // this variant can cause duplicate items in the priority queue
```

Source code: `ch4_05_dijkstra.cpp/java`

[8]The usual implementation of Dijkstra's (e.g. see [7, 38, 8]) requires `heapDecreaseKey` operation in binary heap DS that is not supported by built-in priority queue in C++ STL or Java API. Dijkstra's implementation variant discussed in this section uses only two basic priority queue operations: `enqueue` and `dequeue`.

[9]The operation: `relax(u, v, w_u_v)` sets `dist[v] = min(dist[v], dist[u] + w_u_v)`.

In Figure 4.17, we show a step by step example of running this Dijkstra's implementation variant on a small graph and $s = 2$. Take a careful look at the content of `pq` at each step.

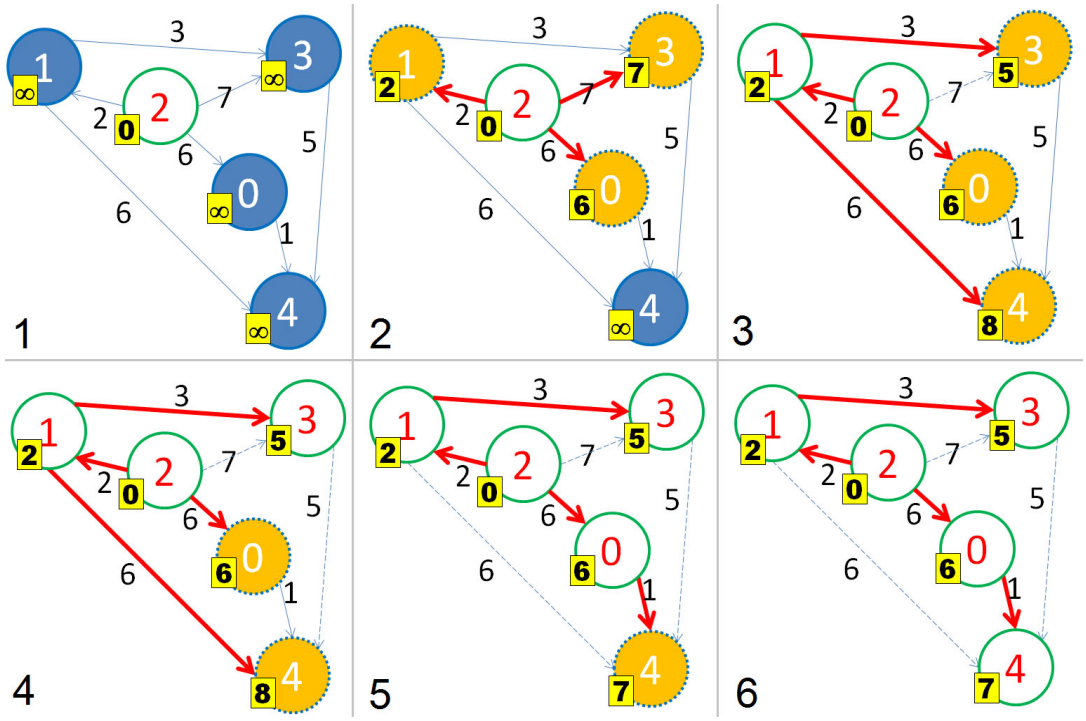


Figure 4.17: Dijkstra Animation on a Weighted Graph (from UVa 341 [47])

1. At the beginning, only `dist[s] = dist[2] = 0`, `priority_queue` pq is {(0,2)}.
2. Dequeue vertex information pair (0,2) from pq. Relax edges incident to vertex 2 to get `dist[0] = 6`, `dist[1] = 2`, and `dist[3] = 7`. Now pq contains {(2,1), (6,0), (7,3)}.
3. Among the unprocessed pairs in pq, (2,1) is in the front of pq. We dequeue (2,1) and relax edges incident to vertex 1 to get `dist[3] = min(dist[3], dist[1] + weight(1,3)) = min(7, 2+3) = 5` and `dist[4] = 8`. Now pq contains {(5,3), (6,0), (7,3), (8,4)}. See that we have 2 entries of vertex 3 in our pq with increasing distance from source s. We do not immediately delete the inferior pair (7,3) from the pq and rely on future iterations of our Dijkstra's variant to correctly pick the one with minimal distance later, which is pair (5,3). This is called 'lazy deletion'.
4. We dequeue (5,3) and try to do `relax(3,4,5)`, i.e. 5+5 = 10. But `dist[4] = 8` (from path 2-1-4), so `dist[4]` is unchanged. Now pq contains {(6,0), (7,3), (8,4)}.
5. We dequeue (6,0) and do `relax(0,4,1)`, making `dist[4] = 7` (the shorter path from 2 to 4 is now 2-0-4 instead of 2-1-4). Now pq contains {(7,3), (7,4), (8,4)} with 2 entries of vertex 4. This is another case of 'lazy deletion'.
6. Now, (7,3) can be ignored as we know that its `d > dist[3]` (i.e. $7 > 5$). This iteration 6 is where the actual deletion of the inferior pair (7,3) is executed rather than iteration 3 earlier. By deferring it until iteration 6, the inferior pair (7,3) is now located in the easy position for the standard $O(\log n)$ deletion in the min heap: At the root of the min heap, i.e. the front of the priority queue.
7. Then (7,4) is processed as before but nothing change. Now pq contains only {(8,4)}.
8. Finally (8,4) is ignored again as its `d > dist[4]` (i.e. $8 > 7$). This Dijkstra's implementation variant stops here as the pq is now empty.

Sample Application: UVa 11367 - Full Tank?

Abridged problem description: Given a connected weighted graph *length* that stores the road length between E pairs of cities i and j ($1 \leq V \leq 1000, 0 \leq E \leq 10000$), the price $p[i]$ of fuel at each city i, and the fuel tank capacity c of a car ($1 \leq c \leq 100$), determine the cheapest trip cost from starting city s to ending city e using a car with fuel capacity c. All cars use one unit of fuel per unit of distance and start with an empty fuel tank.

With this problem, we want to discuss the importance of *graph modeling*. The explicitly given graph in this problem is a weighted graph of the road network. However, we cannot solve this problem with just this graph. This is because the state[10] of this problem requires not just the current location (city) but also the fuel level at that location. Otherwise, we cannot determine whether the car has enough fuel to make a trip along a certain road (because we cannot refuel in the middle of the road). Therefore, we use a pair of information to represent the state: $(location, fuel)$ and by doing so, the total number of vertices of the modified graph *explodes* from just 1000 vertices to $1000 \times 100 = 100000$ vertices. We call the modified graph: 'State-Space' graph.

In the State-Space graph, the source vertex is state $(s, 0)$—at starting city s with empty fuel tank and the target vertices are states (e, any)—at ending city e with any level of fuel between $[0..c]$. There are two types of edge in the State-Space graph: 0-weighted edge that goes from vertex $(x, fuel_x)$ to vertex $(y, fuel_x - length(x, y))$ if the car has sufficient fuel to travel from vertex x to vertex y, and the $p[x]$-weighted edge that goes from vertex $(x, fuel_x)$ to vertex $(x, fuel_x + 1)$ if the car can refuel at vertex x by one unit of fuel (note that the fuel level cannot exceed the fuel tank capacity c). Now, running Dijkstra's on this State-Space graph gives us the solution for this problem (also see Section 8.2.3 for further discussions).

Exercise 4.4.3.1: The modified Dijkstra's implementation variant above may be different from what you learn from other books (e.g. [7, 38, 8]). Analyze if this variant still runs in $O((V+E) \log V)$ on various types of weighted graphs (also see the next **Exercise 4.4.3.2***)?

Exercise 4.4.3.2*: Construct a graph that has negative weight edges but no negative cycle that can significantly slow down this Dijkstra's implementation!

Exercise 4.4.3.3: The sole reason why this variant allows duplicate vertices in the priority queue is so that it can use built-in priority queue library as it is. There is another alternative implementation variant that also has minimal coding. It uses `set`. Implement this variant!

Exercise 4.4.3.4: The source code shown above uses `priority_queue< ii, vector<ii>, greater<ii> > pq;` to sort pairs of integers by increasing distance from source s. How can we achieve the same effect without defining comparison operator for the `priority_queue`? Hint: We have used similar trick with Kruskal's algorithm implementation in Section 4.3.2.

Exercise 4.4.3.5: In **Exercise 4.4.2.2**, we have seen a way to speed up the solution of a shortest paths problem if you are given both the source and the destination vertices. Can the same speedup trick be used for all kinds of weighted graph?

Exercise 4.4.3.6: The graph modeling for UVa 11367 above transform the SSSP problem on weighted graph into SSSP problem on weighted *State-Space* graph. Can we solve this problem with DP? If we can, why? If we cannot, why not? Hint: Read Section 4.7.1.

---

[10]Recall: State is a subset of parameters of the problem that can succinctly describes the problem.

4.4.4 SSSP on Graph with Negative Weight Cycle

If the input graph has negative edge weight, typical Dijkstra's implementation (e.g. [7, 38, 8]) can produces wrong answer. However, Dijkstra's implementation variant shown in Section 4.4.3 above will work just fine, albeit slower. Try it on the graph in Figure 4.18.

This is because Dijkstra's implementation variant will keep inserting new vertex information pair into the priority queue every time it does a relax operation. So, if the graph has no negative weight *cycle*, the variant will keep propagating the shortest path distance information until there is no more possible relaxation (which implies that all shortest paths from the source have been found). However, when given a graph with negative weight *cycle*, the variant—if implemented as shown in Section 4.4.3 above—will be trapped in an infinite loop.

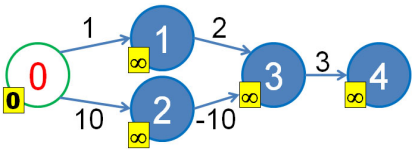


Figure 4.18: -ve Weight

Example: See the graph in Figure 4.19. Path 1-2-1 is a negative cycle. The weight of this cycle is 15 + (-42) = -27.

To solve the SSSP problem in the potential presence of negative weight *cycle(s)*, the more generic (but slower) Bellman Ford's algorithm must be used. This algorithm was invented by Richard Ernest *Bellman* (the pioneer of DP techniques) and Lester Randolph *Ford*, Jr (the same person who invented Ford Fulkerson's method in Section 4.6.2). The main idea of this algorithm is simple: Relax all E edges (in arbitrary order) V-1 times!

Initially `dist[s] = 0`, the base case. If we relax an edge $s \rightarrow u$, then `dist[u]` will have the correct value. If we then relax an edge $u \rightarrow v$, then `dist[v]` will also have the correct value. If we have relaxed all E edges V-1 times, then the shortest path from the source vertex to the furthest vertex from the source (which will be a simple path with V-1 edges) should have been correctly computed. The main part of Bellman Ford's code is simpler than BFS and Dijsktra's code:

```
vi dist(V, INF); dist[s] = 0;
for (int i = 0; i < V - 1; i++)                  // relax all E edges V-1 times
  for (int u = 0; u < V; u++)    // these two loops = O(E), overall O(VE)
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];              // record SP spanning here if needed
      dist[v.first] = min(dist[v.first], dist[u] + v.second);    // relax
    }
```

The complexity of Bellman Ford's algorithm is $O(V^3)$ if the graph is stored as an Adjacency Matrix or $O(VE)$ if the graph is stored as an Adjacency List. This is simply because if we use Adjacency Matrix, we need $O(V^2)$ to enumerate all the edges in our graph. Both time complexities are (much) slower compared to Dijkstra's. However, the way Bellman Ford's works ensure that it will never be trapped in an infinite loop even if the given graph has negative cycle. In fact, Bellman Ford's algorithm can be used to detect *the presence* of negative cycle (e.g. UVa 558 - Wormholes) although such SSSP problem is ill-defined.

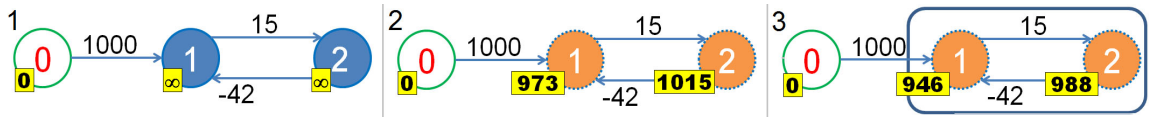


Figure 4.19: Bellman Ford's can detect the presence of negative cycle (from UVa 558 [47])

In **Exercise 4.4.4.1**, we prove that after relaxing all E edges V-1 times, the SSSP problem should have been solved, i.e. we cannot relax any more edge. As the corollary: If we can still relax an edge, there must be a negative cycle in our weighted graph.

For example, in Figure 4.19—left, we see a simple graph with a negative cycle. After 1 pass, `dist[1] = 973` and `dist[2] = 1015` (middle). After V-1 = 2 passes, `dist[1] = 946` and `dist[2] = 988` (right). As there is a negative cycle, we can still do this again (and again), i.e. we can still relax `dist[2] = 946+15 = 961`. This is lower than the current value of `dist[2] = 988`. The presence of a negative cycle causes the vertices reachable from this negative cycle to have ill-defined shortest paths information. This is because one can simply traverse this negative cycle infinite number of times to make all reachable vertices from this negative cycle to have negative infinity shortest paths information. The code to check for negative cycle is simple:

```
// after running the O(VE) Bellman Ford's algorithm shown above
  bool hasNegativeCycle = false;
  for (int u = 0; u < V; u++)                       // one more pass to check
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (dist[v.first] > dist[u] + v.second)  // if this is still possible
        hasNegativeCycle = true;              // then negative cycle exists!
    }
  printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");
```

In programming contests, the slowness of Bellman Ford's and its negative cycle detection feature causes it to be used only to solve the SSSP problem on *small* graph which is *not guaranteed* to be free from negative weight cycle.

Exercise 4.4.4.1: Why just by relaxing all E edges of our weighted graph $V - 1$ times, we will have the correct SSSP information? Prove it!

Exercise 4.4.4.2: The worst case time complexity of $O(VE)$ is too large in practice. For most cases, we can actually stop Bellman Ford's (much) earlier. Suggest a simple improvement to the given code above to make Bellman Ford's *usually* runs faster than $O(VE)$!

Exercise 4.4.4.3*: A known improvement for Bellman Ford's (especially among Chinese programmers) is the SPFA (Shortest Path Faster Algorithm). Study Section 9.30!

Visualization: `www.comp.nus.edu.sg/~stevenha/visualization/sssp.html`

Source code: `ch4_06_bellman_ford.cpp/java`

Programming Exercises related to Single-Source Shortest Paths:

- On Unweighted Graph: BFS, Easier
  1. UVa 00336 - A Node Too Far (discussed in this section)
  2. UVa 00383 - Shipping Routes (simple SSSP solvable with BFS, use mapper)
  3. *UVa 00388 - Galactic Import* (key idea: we want to minimize planet movements because every edge used decreases value by 5%)
  4. **UVa 00429 - Word Transformation *** (each word is a vertex, connect 2 words with an edge if differ by 1 letter)
  5. UVa 00627 - The Net (also print the path, see discussion in this section)
  6. UVa 00762 - We Ship Cheap (simple SSSP solvable with BFS, use mapper)
  7. **UVa 00924 - Spreading the News *** (the spread is like BFS traversal)
  8. *UVa 01148 - The mysterious X network* (LA 3502, SouthWesternEurope05, single source, single target, shortest path problem but exclude endpoints)
  9. UVa 10009 - All Roads Lead Where? (simple SSSP solvable with BFS)
  10. UVa 10422 - Knights in FEN (solvable with BFS)
  11. UVa 10610 - Gopher and Hawks (solvable with BFS)
  12. **UVa 10653 - Bombs; NO they ... *** (efficient BFS implementation)
  13. UVa 10959 - The Party, Part I (SSSP from source 0 to the rest)
- On Unweighted Graph: BFS, Harder
  1. **UVa 00314 - Robot *** (state: (position, direction), transform input graph)
  2. UVa 00532 - Dungeon Master (3-D BFS)
  3. *UVa 00859 - Chinese Checkers* (BFS)
  4. *UVa 00949 - Getaway* (interesting graph data structure twist)
  5. UVa 10044 - Erdos numbers (the input parsing part is troublesome; if you encounter difficulties with this, see Section 6.2)
  6. UVa 10067 - Playing with Wheels (implicit graph in problem statement)
  7. UVa 10150 - Doublets (BFS state is string!)
  8. UVa 10977 - Enchanted Forest (BFS with blocked states)
  9. UVa 11049 - Basic Wall Maze (some restricted moves, print the path)
  10. **UVa 11101 - Mall Mania *** (multi-sources BFS from m1, get minimum at border of m2)
  11. UVa 11352 - Crazy King (filter the graph first, then it becomes SSSP)
  12. UVa 11624 - Fire (multi-sources BFS)
  13. UVa 11792 - Krochanska is Here (be careful with the 'important station')
  14. **UVa 12160 - Unlock the Lock *** (LA 4408, KualaLumpur08, Vertices = The numbers; Link two numbers with an edge if we can use button push to transform one into another; use BFS to get the answer)
- On Weighted Graph: Dijkstra's, Easier
  1. **UVa 00929 - Number Maze *** (on a 2D maze/implicit graph)
  2. ***UVa 01112 - Mice and Maze **** (LA 2425, SouthwesternEurope01, run Dijkstra's from destination)
  3. UVa 10389 - Subway (use basic geometry skill to build the weighted graph, then run Dijkstra's)
  4. **UVa 10986 - Sending email *** (straightforward Dijkstra's application)

- On Weighted Graph: Dijkstra's, Harder
  1. UVa 01202 - Finding Nemo (LA 3133, Beijing04, SSSP, Dijkstra's on grid: treat each cell as a vertex; the idea is simple but one should be careful with the implementation)
  2. UVa 10166 - Travel (this can be modeled as a shortest paths problem)
  3. *UVa 10187 - From Dusk Till Dawn* (special cases: start = destination: 0 litre; starting or destination city not found or destination city not reachable from starting city: no route; the rest: Dijkstra's)
  4. UVa 10278 - Fire Station (Dijkstra's from fire stations to all intersections; need pruning to pass the time limit)
  5. *UVa 10356 - Rough Roads* (we can attach one extra information to each vertex: whether we come to that vertex using cycle or not; then, run Dijkstra's to solve SSSP on this modified graph)
  6. UVa 10603 - Fill (state: (a, b, c), source: (0, 0, c), 6 possible transitions)
  7. **UVa 10801 - Lift Hopping *** (model the graph carefully!)
  8. *UVa 10967 - The Great Escape* (model the graph; shortest path)
  9. *UVa 11338 - Minefield* (it seems that the test data is weaker than what the problem description says ($n \leq 10000$); we use $O(n^2)$ loop to build the weighted graph and runs Dijkstra's without getting TLE)
  10. UVa 11367 - Full Tank? (discussed in this section)
  11. UVa 11377 - Airport Setup (model the graph carefully: A city to other city with no airport has edge weight 1. A city to other city with airport has edge weight 0. Do Dijkstra's from source. If the start and end city are the same and has no airport, the answer should be 0.)
  12. **UVa 11492 - Babel *** (graph modeling; each word is a vertex; connect two vertices with an edge if they share common language and have different 1st char; connect a source vertex to all words that belong to start language; connect all words that belong to finish language to sink vertex; we can transfer vertex weight to edge weight; then SSSP from source vertex to sink vertex)
  13. UVa 11833 - Route Change (stop Dijkstra's at service route path plus some modification)
  14. ***UVa 12047 - Highest Paid Toll ** (clever usage of Dijkstra's; run Dijkstra's from source and from destination; try all edge (u, v) if $dist[source][u] + weight(u, v) + dist[v][destination] \leq p$; record the largest edge weight found)
  15. *UVa 12144 - Almost Shortest Path* (Dijkstra's; store multiple predecessors)
  16. IOI 2011 - Crocodile (can be modeled as an SSSP problem)
- SSSP on Graph with Negative Weight Cycle (Bellman Ford's)
  1. **UVa 00558 - Wormholes *** (checking the existence of negative cycle)
  2. ***UVa 10449 - Traffic ** (find the minimum weight path, which may be negative; be careful: ∞ + negative weight is lower than ∞!)
  3. **UVa 10557 - XYZZY *** (check 'positive' cycle, check connectedness!)
  4. UVa 11280 - Flying to Fredericton (modified Bellman Ford's)

4.5 All-Pairs Shortest Paths

4.5.1 Overview and Motivation

Motivating Problem: Given a connected, weighted graph G with $V \leq 100$ and two vertices s and d, find the maximum possible value of `dist[s][i] + dist[i][d]` over all possible $i \in [0 \ldots V-1]$. This is the key idea to solve UVa 11463 - Commandos. However, what is the best way to implement the solution code for this problem?

This problem requires the shortest path information from all possible sources (all possible vertices) of G. We can make V calls of Dijkstra's algorithm that we have learned earlier in Section 4.4.3 above. However, can we solve this problem in a *shorter way*—in terms of code length? The answer is yes. If the given weighted graph has $V \leq 400$, then there is another algorithm that is *simpler to code*.

Load the small graph into an Adjacency Matrix and then run the following four-liner code with three nested loops shown below. When it terminates, `AdjMat[i][j]` will contain the shortest path distance between two pair of vertices i and j in G. The original problem (UVa 11463 above) now becomes easy.

```
// inside int main()
  // precondition: AdjMat[i][j] contains the weight of edge (i, j)
  // or INF (1B) if there is no such edge
  // AdjMat is a 32-bit signed integer array
  for (int k = 0; k < V; k++)        // remember that loop order is k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMat[i][j] = min(AdjMat[i][j], AdjMat[i][k] + AdjMat[k][j]);
```

Source code: `ch4_07_floyd_warshall.cpp/java`

This algorithm is called Floyd Warshall's algorithm, invented by Robert W *Floyd* [19] and Stephen *Warshall* [70]. Floyd Warshall's is a DP algorithm that clearly runs in $O(V^3)$ due to its 3 nested loops[11]. Therefore, it can only be used for graph with $V \leq 400$ in programming contest setting. In general, Floyd Warshall's solves another classical graph problem: The All-Pairs Shortest Paths (APSP) problem. It is an alternative algorithm (for small graphs) compared to calling SSSP algorithm multiple times:

1. V calls of $O((V+E)\log V)$ Dijkstra's $= O(V^3 \log V)$ if $E = O(V^2)$.
2. V calls of $O(VE)$ Bellman Ford's $= O(V^4)$ if $E = O(V^2)$.

In programming contest setting, Floyd Warshall's main attractiveness is basically its implementation speed—four short lines only. If the given graph is small ($V \leq 400$), do not hesitate to use this algorithm—even if you only need a solution for the SSSP problem.

Exercise 4.5.1.1: Is there any specific reason why `AdjMat[i][j]` must be set to 1B (10^9) to indicate that there is no edge between 'i' to 'j'? Why don't we use $2^{31}-1$ (MAX_INT)?

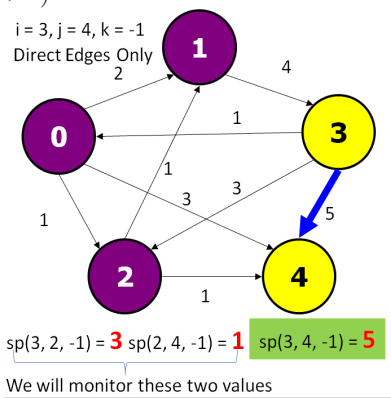
Exercise 4.5.1.2: In Section 4.4.4, we differentiate graph with negative weight edges but no negative cycle and graph with negative cycle. Will this short Floyd Warshall's algorithm works on graph with negative weight and/or negative cycle? Do some experiment!

---

[11] Floyd Warshall's must use Adjacency Matrix so that the weight of edge (i, j) can be accessed in $O(1)$.

4.5.2 Explanation of Floyd Warshall's DP Solution

We provide this section for the benefit of readers who are interested to know why Floyd Warshall's works. This section can be skipped if you just want to use this algorithm per se. However, examining this section can further strengthen your DP skill. Note that there are graph problems that have no classical algorithm yet and must be solved with DP techniques (see Section 4.7.1).

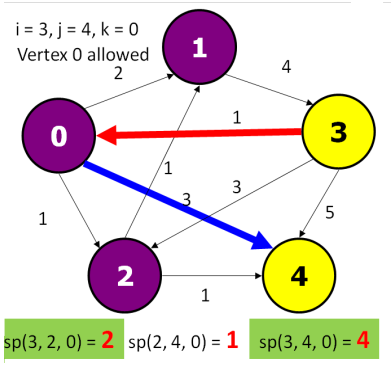


The current content of Adjacency Matrix D at **k = -1**

k = -1	0	1	2	3	4
0	0	2	1	∞	3
1	∞	0	∞	4	∞
2	∞	1	0	∞	1
3	1	∞	3	0	**5**
4	∞	∞	∞	∞	0

Figure 4.20: Floyd Warshall's Explanation 1

The basic idea behind Floyd Warshall's is to gradually allow the usage of intermediate vertices (vertex `[0..k]`) to form the shortest paths. We denote the shortest path from vertex i to vertex j using only intermediate vertices `[0..k]` as `sp(i,j,k)`. Let the vertices be labeled from 0 to V-1. We start with direct edges only when $k = -1$, i.e. `sp(i,j,-1)` = weight of edge (i, j). Then, we find shortest paths between any two vertices with the help of restricted intermediate vertices from vertex `[0..k]`. In Figure 4.20, we want to find `sp(3,4,4)`—the shortest path from vertex 3 to vertex 4, using any intermediate vertex in the graph (vertex `[0..4]`). The eventual shortest path is path 3-0-2-4 with cost 3. But how to reach this solution? We know that by using only direct edges, `sp(3,4,-1) = 5`, as shown in Figure 4.20. The solution for `sp(3,4,4)` will *eventually* be reached from `sp(3,2,2)+sp(2,4,2)`. But with using only direct edges, `sp(3,2,-1)+sp(2,4,-1) = 3+1 = 4` is still > 3.

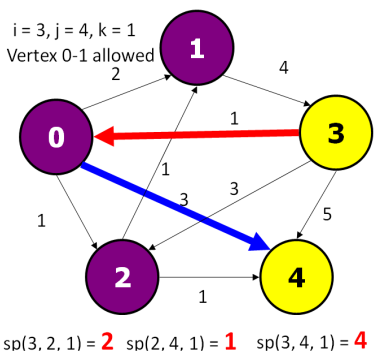


The current content of Adjacency Matrix D at **k = 0**

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

Figure 4.21: Floyd Warshall's Explanation 2

Floyd Warshall's then gradually allow $k = 0$, then $k = 1$, $k = 2 \ldots$, up to $k = V$-1. When we allow $k = 0$, i.e. vertex 0 can now be used as an intermediate vertex, then `sp(3,4,0)` is reduced as `sp(3,4,0) = sp(3,0,-1) + sp(0,4,-1) = 1+3 = 4`, as shown in Figure 4.21. Note that with $k = 0$, `sp(3,2,0)`—which we will need later—also drop from 3 to `sp(3,0,-1) + sp(0,2,-1) = 1+1 = 2`. Floyd Warshall's will process `sp(i,j,0)` for all other pairs considering only vertex 0 as the intermediate vertex but there is only one more change: `sp(3,1,0)` from ∞ down to 3.

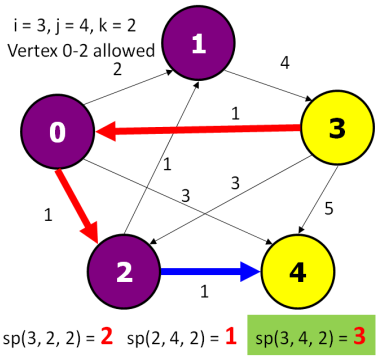


The current content of Adjacency Matrix D at **k = 1**

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

Figure 4.22: Floyd Warshall's Explanation 3

When we allow $k = 1$, i.e. vertex 0 and 1 can now be used as intermediate vertices, then it happens that there is no change to `sp(3,2,1)`, `sp(2,4,1)`, nor to `sp(3,4,1)`. However, two other values change: `sp(0,3,1)` and `sp(2,3,1)` as shown in Figure 4.22 but these two values will not affect the final computation of the shortest path between vertex 3 and 4.



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

Figure 4.23: Floyd Warshall's Explanation 4

When we allow $k = 2$, i.e. vertex 0, 1, and 2 now can be used as the intermediate vertices, then `sp(3,4,2)` is reduced again as `sp(3,4,2) = sp(3,2,2)+sp(2,4,2) = 2+1 = 3` as shown in Figure 4.23. Floyd Warshall's repeats this process for $k = 3$ and finally $k = 4$ but `sp(3,4,4)` remains at 3 and this is the final answer.

Formally, we define Floyd Warshall's DP recurrences as follow. Let $D^k_{i,j}$ be the shortest distance from i to j with only `[0..k]` as intermediate vertices. Then, Floyd Warshall's base case and recurrence are as follow:

$D^{-1}_{i,j} = weight(i, j)$. This is the base case when we do not use any intermediate vertices.
$D^k_{i,j} = min(D^{k-1}_{i,j}, D^{k-1}_{i,k} + D^{k-1}_{k,j}) = min(\text{not using vertex } k, \text{using vertex } k)$, for $k \geq 0$.

This DP formulation must be filled layer by layer (by increasing k). To fill out an entry in the table k, we make use of the entries in the table k-1. For example, to calculate $D^2_{3,4}$, (row 3, column 4, in table $k = 2$, index start from 0), we look at the minimum of $D^1_{3,4}$ or the sum of $D^1_{3,2} + D^1_{2,4}$ (see Table 4.3). The naïve implementation is to use a 3-dimensional matrix `D[k][i][j]` of size $O(V^3)$. However, since to compute layer k we only need to know the values from layer k-1, we can drop dimension k and compute `D[i][j]` 'on-the-fly' (the space saving trick discussed in Section 3.5.1). Thus, Floyd Warshall's algorithm just need $O(V^2)$ space although it still runs in $O(V^3)$.

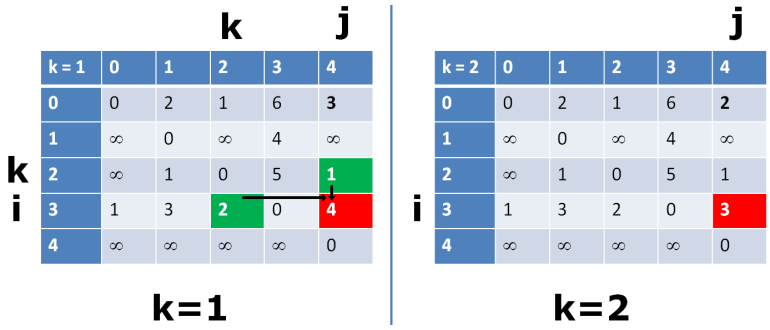


Table 4.3: Floyd Warshall's DP Table

4.5.3 Other Applications

The main purpose of Floyd Warshall's is to solve the APSP problem. However, Floyd Warshall's is frequently used in other problems too, as long as the input graph is small. Here we list down several problem variants that are also solvable with Floyd Warshall's.

Solving the SSSP Problem on a Small Weighted Graph

If we have the All-Pairs Shortest Paths (APSP) information, we also know the Single-Source Shortest Paths (SSSP) information from any possible source. If the given weighted graph is small $V \leq 400$, it may be beneficial, in terms of coding time, to use the four-liner Floyd Warshall's code rather than the longer Dijkstra's algorithm.

Printing the Shortest Paths

A common issue encountered by programmers who use the four-liner Floyd Warshall's without understanding how it works is when they are asked to print the shortest paths too. In BFS/Dijkstra's/Bellman Ford's algorithms, we just need to remember the shortest paths spanning tree by using a 1D `vi p` to store the parent information for each vertex. In Floyd Warshall's, we need to store a 2D parent matrix. The modified code is shown below.

```
// inside int main()
  // let p be a 2D parent matrix, where p[i][j] is the last vertex before j
  // on a shortest path from i to j, i.e. i -> ... -> p[i][j] -> j
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      p[i][j] = i;                          // initialize the parent matrix
  for (int k = 0; k < V; k++)
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++) // this time, we need to use if statement
        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 the parent matrix
        }
//--------------------------------------------------------------------------
// when we need to print the shortest paths, we can call the method below:
void printPath(int i, int j) {
  if (i != j) printPath(i, p[i][j]);
  printf(" %d", j);
}
```

Transitive Closure (Warshall's Algorithm)

Stephen *Warshall* [70] developed an algorithm for the Transitive Closure problem: Given a graph, determine if vertex i is connected to j, directly *or indirectly*. This variant uses logical bitwise operators which is (much) faster than arithmetic operators. Initially, `AdjMat[i][j]` contains 1 (`true`) if vertex i is *directly* connected to vertex j, 0 (`false`) otherwise. After running $O(V^3)$ Warshall's algorithm below, we can check if any two vertices i and j are directly or indirectly connected by checking `AdjMat[i][j]`.

```
for (int k = 0; k < V; k++)
  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      AdjMat[i][j] |= (AdjMat[i][k] & AdjMat[k][j]);
```

Minimax and Maximin (Revisited)

We have seen the minimax (and maximin) path problem earlier in Section 4.3.4. The solution using Floyd Warshall's is shown below. First, initialize `AdjMat[i][j]` to be the weight of edge `(i,j)`. This is the default minimax cost for two vertices that are directly connected. For pair `i-j` without any direct edge, set `AdjMat[i][j] = INF`. Then, we try all possible intermediate vertex k. The minimax cost `AdjMat[i][j]` is the minimum of either (itself) or (the maximum between `AdjMat[i][k]` or `AdjMat[k][j]`). However, this approach can only be used if the input graph is small enough ($V \leq 400$).

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

Finding the (Cheapest/Negative) Cycle

In Section 4.4.4, we have seen how Bellman Ford's terminates after $O(VE)$ steps regardless of the type of input graph (as it relax all E edges at most V-1 times) and how Bellman Ford's can be used to check if the given graph has negative cycle. Floyd Warshall's also terminates after $O(V^3)$ steps regardless of the type of input graph. This feature allows Floyd Warshall's to be used to detect whether the (small) graph has a cycle, a negative cycle, and even finding the cheapest (non-negative) cycle among all possible cycles (the girth of the graph).

To do this, we initially set the *main diagonal* of the Adjacency Matrix to have a very large value, i.e. `AdjMat[i][i] = INF (1B)`. Then, we run the $O(V^3)$ Floyd Warshall's algorithm. Now, we check the value of `AdjMat[i][i]`, which now means the shortest cyclic path weight starting from vertex `i` that goes through up to V-1 other intermediate vertices and returns back to `i`. If `AdjMat[i][i]` is no longer INF for any `i` $\in$ `[0..V-1]`, then we have a cycle. The smallest non-negative `AdjMat[i][i]` $\forall$`i` $\in$ `[0..V-1]` is the *cheapest* cycle. If `AdjMat[i][i] < 0` for any `i` $\in$ `[0..V-1]`, then we have a *negative* cycle because if we take this cyclic path one more time, we will get an even shorter 'shortest' path.

Finding the Diameter of a Graph

The diameter of a graph is defined as the maximum shortest path distance between any pair of vertices of that graph. To find the diameter of a graph, we first find the shortest path

between each pair of vertices (i.e. the APSP problem). The maximum distance found is the diameter of the graph. UVa 1056 - Degrees of Separation, which is an ICPC World Finals problem in 2006, is precisely this problem. To solve this problem, we can first run an $O(V^3)$ Floyd Warshall's to compute the required APSP information. Then, we can figure out what is the diameter of the the graph by finding the maximum value in the `AdjMat` in $O(V^2)$. However, we can only do this for a small graph with $V \leq 400$.

Finding the SCCs of a Directed Graph

In Section 4.2.1, we have learned how the $O(V + E)$ Tarjan's algorithm can be used to identify the SCCs of a directed graph. However, the code is a bit long. If the input graph is small (e.g. UVa 247 - Calling Circles, UVa 1229 - Sub-dictionary, UVa 10731 - Test), we can also identify the SCCs of the graph in $O(V^3)$ using Warshall's transitive closure algorithm and then use the following check: To find all members of an SCC that contains vertex `i`, check all other vertices `j` ∈ `[0..V-1]`. If `AdjMat[i][j] && AdjMat[j][i]` is true, then vertex `i` and `j` belong to the same SCC.

Exercise 4.5.3.1: How to find the transitive closure of a graph with $V \leq 1000, E \leq 100000$? Suppose that there are only Q ($1 \leq 100 \leq Q$) transitive closure queries for this problem in form of this question: Is vertex u connected to vertex v, directly or indirectly? What if the input graph is *directed*? Does this directed property simplify the problem?

Exercise 4.5.3.2*: Solve the *maximin* path problem using Floyd Warshall's!

Exercise 4.5.3.3: Arbitrage is the trading of one currency for another with the hopes of taking advantage of small differences in conversion rates among several currencies in order to achieve a profit. For example (UVa 436 - Arbitrage II): If 1.0 United States dollar (USD) buys 0.5 British pounds (GBP), 1.0 GBP buys 10.0 French francs (FRF[12]), and 1.0 FRF buys 0.21 USD, then an arbitrage trader can start with 1.0 USD and buy $1.0 \times 0.5 \times 10.0 \times 0.21 = 1.05$ USD thus earning a profit of 5 percent. This problem is actually a problem of finding a *profitable cycle*. It is akin to the problem of finding cycle with Floyd Warshall's shown in this section. Solve the arbitrage problem using Floyd Warshall's!

Remarks About Shortest Paths in Programming Contests

All three algorithms discussed in the past two sections: Dijkstra's, Bellman Ford's, and Floyd Warshall's are used to solve the *general case* of shortest paths (SSSP or APSP) problems on weighted graphs. Out of these three, the $O(VE)$ Bellman Ford's is rarely used in programming contests due to its high time complexity. It is only useful if the problem author gives a 'reasonable size' graph with negative cycle. For general cases, (our modified) $O((V+E) \log V)$ Dijkstra's implementation variant is the best solution for the SSSP problem for 'reasonable size' weighted graph without negative cycle. However, when the given graph is small ($V \leq 400$)—which happens many times, it is clear from this section that the $O(V^3)$ Floyd Warshall's is the best way to go.

One possible reason on why Floyd Warshall's algorithm is quite popular in programming contests is because sometimes the problem author includes shortest paths as the *sub-problem* of the main, (much) more complex, problem. To make the problem still doable during contest time, the problem author purposely sets the input size to be small so that the shortest paths

[12]At the moment (2013), France actually uses Euro as its currency.