Chapter 4

Graph

Everyone is on average ≈ six steps away from any other person on Earth
— **Stanley Milgram - the Six Degrees of Separation experiment in 1969, [64]**

4.1 Overview and Motivation

Many real-life problems can be classified as graph problems. Some have efficient solutions. Some do not have them yet. In this relatively big chapter with lots of figures, we discuss graph problems that commonly appear in programming contests, the algorithms to solve them, and the *practical* implementations of these algorithms. We cover topics ranging from basic graph traversals, minimum spanning trees, single-source/all-pairs shortest paths, network flows, and discuss graphs with special properties.

In writing this chapter, we assume that the readers are *already* familiar with the graph terminologies listed in Table 4.1. If you encounter any unfamiliar term, please read other reference books like [7, 58] (or browse the Internet) and search for that particular term.

Vertices/Nodes	Edges	Set V; size $\|V\|$	Set E; size $\|E\|$	Graph $G(V, E)$
Un/Weighted	Un/Directed	Sparse	Dense	In/Out Degree
Path	Cycle	Isolated	Reachable	Connected
Self-Loop	Multiple Edges	Multigraph	Simple Graph	Sub-Graph
DAG	Tree/Forest	Eulerian	Bipartite	Complete

Table 4.1: List of Important Graph Terminologies

We also assume that the readers have read various ways to represent graph information that have been discussed earlier in Section 2.4.1. That is, we will directly use the terms like: Adjacency Matrix, Adjacency List, Edge List, and implicit graph without redefining them. Please revise Section 2.4.1 if you are not familiar with these graph data structures.

Our research so far on graph problems in recent ACM ICPC (Asia) regional contests reveals that there is at least one (and possibly more) graph problem(s) in an ICPC problem set. However, since the range of graph problems is so big, each graph problem only has a small probability of appearance. So the question is "Which ones do we have to focus on?". In our opinion, there is no clear answer for this question. If you want to do well in ACM ICPC, you have no choice but to study and master all these materials.

For IOI, the syllabus [20] restricts IOI tasks to a subset of material mentioned in this chapter. This is logical as high school students competing in IOI are not expected to be well-versed with too many problem-specific algorithms. To assist the readers aspiring to take part in the IOI, we will mention whether a particular section in this chapter is currently outside the syllabus.

4.2 Graph Traversal

4.2.1 Depth First Search (DFS)

Depth First Search—abbreviated as DFS—is a simple algorithm for traversing a graph. Starting from a distinguished source vertex, DFS will traverse the graph 'depth-first'. Every time DFS hits a branching point (a vertex with more than one neighbors), DFS will choose one of the unvisited neighbor(s) and visit this neighbor vertex. DFS repeats this process and goes deeper until it reaches a vertex where it cannot go any deeper. When this happens, DFS will 'backtrack' and explore another unvisited neighbor(s), if any.

This graph traversal behavior can be implemented easily with the recursive code below. Our DFS implementation uses the help of a *global* vector of integers: `vi dfs_num` to distinguish the state of each vertex. For the simplest DFS implementation, we only use `vi dfs_num` to distinguish between 'unvisited' (we use a constant value `UNVISITED = -1`) and 'visited' (we use another constant value `VISITED = 1`). Initially, all values in `dfs_num` are set to 'unvisited'. We will use `vi dfs_num` for other purposes later. Calling `dfs(u)` starts DFS from a vertex u, marks vertex u as 'visited', and then DFS recursively visits each 'unvisited' neighbor v of u (i.e. edge $u - v$ exists in the graph and `dfs_num[v] == UNVISITED`).

```
typedef pair<int, int> ii; // In this chapter, we will frequently use these
typedef vector<ii> vii; // three data type shortcuts. They may look cryptic
typedef vector<int> vi;   // but they are useful in competitive programming

vi dfs_num;   // global variable, initially all values are set to UNVISITED

void dfs(int u) {     // DFS for normal usage: as graph traversal algorithm
  dfs_num[u] = VISITED;        // important: we mark this vertex as visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) { // default DS: AdjList
    ii v = AdjList[u][j];                 // v is a (neighbor, weight) pair
    if (dfs_num[v.first] == UNVISITED)    // important check to avoid cycle
      dfs(v.first);   // recursively visits unvisited neighbors of vertex u
} }  // for simple graph traversal, we ignore the weight stored at v.second
```

The time complexity of this DFS implementation depends on the graph data structure used. In a graph with V vertices and E edges, DFS runs in $O(V + E)$ and $O(V^2)$ if the graph is stored as Adjacency List and Adjacency Matrix, respectively (see **Exercise 4.2.2.2**).

On the sample graph in Figure 4.1, `dfs(0)`—calling DFS from a starting vertex $u = 0$—will trigger this sequence of visitation: $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4$. This sequence is 'depth-first', i.e. DFS goes to the deepest possible vertex from the start vertex before attempting another branch (there is none in this case).

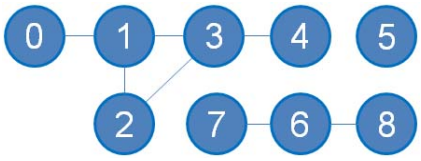


Figure 4.1: Sample Graph

Note that this sequence of visitation depends very much on the way we order the neighbors of a vertex[1], i.e. the sequence $0 \rightarrow 1 \rightarrow 3 \rightarrow 2$ (backtrack to 3) $\rightarrow 4$ is also a possible visitation sequence.

Also notice that one call of `dfs(u)` will only visit all vertices that are *connected* to vertex u. That is why vertices 5, 6, 7, and 8 in Figure 4.1 remain unvisited after calling `dfs(0)`.

[1]For simplicity, we usually just order the vertices based on their vertex numbers, e.g. in Figure 4.1, vertex 1 has vertex {0, 2, 3} as its neighbor, in that order.

The DFS code shown here is very similar to the recursive backtracking code shown earlier in Section 3.2. If we compare the pseudocode of a typical backtracking code (replicated below) with the DFS code shown above, we can see that the main difference is the flagging of visited vertices (states). Backtracking (automatically) un-flag visited vertices (reset the state to previous state) when the recursion backtracks to allow re-visitation of those vertices (states) from another branch. By not revisiting vertices of a general graph (via `dfs_num` checks), DFS runs in $O(V + E)$, but the time complexity of backtracking is exponential.

```
void backtrack(state) {
  if (hit end state or invalid state)             // we need terminating or
    return;    // pruning condition to avoid cycling and to speed up search
  for each neighbor of this state                    // try all permutation
    backtrack(neighbor);
}
```

Sample Application: UVa 11902 - Dominator

Abridged problem description: Vertex X dominates vertex Y if every path from the a start vertex (vertex 0 for this problem) to Y must go through X. If Y is not reachable from the start vertex then Y does not have any dominator. Every vertex reachable from the start vertex dominates itself. For example, in the graph shown in Figure 4.2, vertex 3 dominates vertex 4 since all the paths from vertex 0 to vertex 4 must pass through vertex 3. Vertex 1 does not dominate vertex 3 since there is a path 0-2-3 that does not include vertex 1. Our task: Given a directed graph, determine the dominators of every vertex.

This problem is about reachability tests from a start vertex (vertex 0). Since the input graph for this problem is small ($V < 100$), we can afford to use the following $O(V \times V^2 = V^3)$ algorithm. Run `dfs(0)` on the input graph to record vertices that are reachable from vertex 0. Then to check which vertices are dominated by vertex X, we (temporarily) turn off all the outgoing edges of vertex X and rerun `dfs(0)`. Now, a vertex Y is not dominated by vertex X if `dfs(0)` initially cannot reach vertex Y or `dfs(0)` can reach vertex Y even after all outgoing edges of vertex X are (temporarily) turned off. Vertex Y is dominated by vertex X otherwise. We repeat this process $\forall X \in [0 \ldots V-1]$.

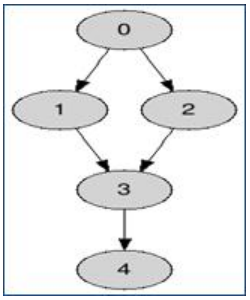


Figure 4.2: UVa 11902

Tips: We do not have to physically delete vertex X from the input graph. We can simply add a statement inside our DFS routine to stop the traversal if it hits vertex X.

4.2.2 Breadth First Search (BFS)

Breadth First Search—abbreviated as BFS—is another graph traversal algorithm. Starting from a distinguished source vertex, BFS will traverse the graph 'breadth-first'. That is, BFS will visit vertices that are direct neighbors of the source vertex (first layer), neighbors of direct neighbors (second layer), and so on, layer by layer.

BFS starts with the insertion of the source vertex s into a queue, then processes the queue as follows: Take out the front most vertex u from the queue, enqueue all unvisited neighbors of u (usually, the neighbors are ordered based on their vertex numbers), and mark them as visited. With the help of the queue, BFS will visit vertex s and all vertices in the connected component that contains s layer by layer. BFS algorithm also runs in $O(V + E)$ and $O(V^2)$

on a graph represented using an Adjacency List and Adjacency Matrix, respectively (again, see **Exercise 4.2.2.2**).

Implementing BFS is easy if we utilize C++ STL or Java API. We use `queue` to order the sequence of visitation and `vector<int>` (or `vi`) to record if a vertex has been visited or not—which at the same time also record the distance (layer number) of each vertex from the source vertex. This distance computation feature is used later to solve a special case of Single-Source Shortest Paths problem (see Section 4.4 and 8.2.3).

```
// inside int main()---no recursion
  vi d(V, INF); d[s] = 0;                 // distance from source s to s is 0
  queue<int> q; q.push(s);                                 // start from source

  while (!q.empty()) {
    int u = q.front(); q.pop();                          // queue: layer by layer!
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                              // for each neighbor of u
      if (d[v.first] == INF) {         // if v.first is unvisited + reachable
        d[v.first] = d[u] + 1;           // make d[v.first] != INF to flag it
        q.push(v.first);             // enqueue v.first for the next iteration
  } } }
```

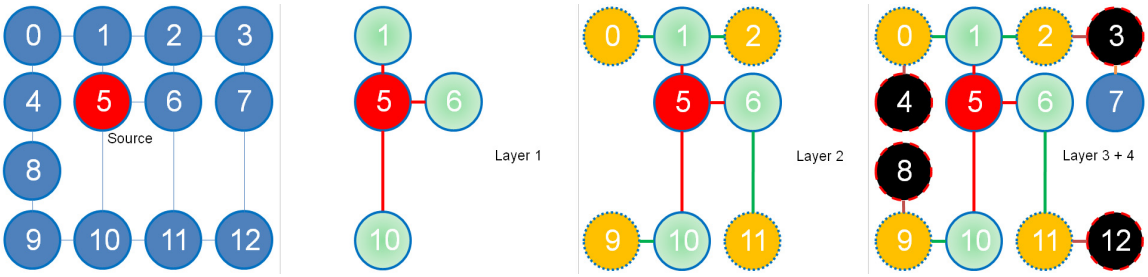


Figure 4.3: Example Animation of BFS

If we run BFS from vertex 5 (i.e. the source vertex $s = 5$) on the connected undirected graph shown in Figure 4.3, we will visit the vertices in the following order:

```
Layer 0:, visit 5
Layer 1:, visit 1, visit 6, visit 10
Layer 2:, visit 0, visit 2, visit 11, visit 9
Layer 3:, visit 4, visit 3, visit 12, visit 8
Layer 4:, visit 7
```

Exercise 4.2.2.1: To show that either DFS or BFS can be used to visit all vertices that are reachable from a source vertex, solve UVa 11902 - Dominator using BFS instead!

Exercise 4.2.2.2: Why do DFS and BFS run in $O(V+E)$ if the graph is stored as Adjacency List and become slower (run in $O(V^2)$) if the graph is stored as Adjacency Matrix? Follow up question: What is the time complexity of DFS and BFS if the graph is stored as Edge List instead? What should we do if the input graph is given as an Edge List and we want to traverse the graph efficiently?

4.2.3 Finding Connected Components (Undirected Graph)

DFS and BFS are not only useful for traversing a graph. They can be used to solve many other graph problems. The first few problems below can be solved with *either* DFS or BFS although some of the last few problems are more suitable for DFS only.

The fact that one single call of `dfs(u)` (or `bfs(u)`) will only visit vertices that are actually connected to u can be utilized to find (and to count the number of) connected components in an *undirected* graph (see further below in Section 4.2.9 for a similar problem on directed graph). We can simply use the following code to restart DFS (or BFS) from one of the remaining unvisited vertices to find the next connected component. This process is repeated until all vertices have been visited and has an overall time complexity of $O(V + E)$.

```
// inside int main()---this is the DFS solution
  numCC = 0;
  dfs_num.assign(V, UNVISITED);    // sets all vertices' state to UNVISITED
  for (int i = 0; i < V; i++)              // for each vertex i in [0..V-1]
    if (dfs_num[i] == UNVISITED)          // if vertex i is not visited yet
      printf("CC %d:", ++numCC), dfs(i), printf("\n");     // 3 lines here!

// For the sample graph in Figure 4.1, the output is like this:
// CC 1: 0 1 2 3 4
// CC 2: 5
// CC 3: 6 7 8
```

Exercise 4.2.3.1: UVa 459 - Graph Connectivity is basically this problem of finding connected components of an undirected graph. Solve it using the DFS solution shown above! However, we can also use Union-Find Disjoint Sets data structure (see Section 2.4.2) or BFS (see Section 4.2.2) to solve this graph problem. How?

4.2.4 Flood Fill - Labeling/Coloring the Connected Components

DFS (or BFS) can be used for other purposes than just finding (and counting the number of) connected components. Here, we show how a *simple tweak* of the $O(V + E)$ `dfs(u)` (we can also use `bfs(u)`) can be used to *label* (also known in CS terminology as 'to color') and count the size of each component. This variant is more famously known as 'flood fill' and usually performed on *implicit* graphs (usually 2D grids).

```
int dr[] = {1,1,0,-1,-1,-1, 0, 1};  // trick to explore an implicit 2D grid
int dc[] = {0,1,1, 1, 0,-1,-1,-1};         // S,SE,E,NE,N,NW,W,SW neighbors

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

Sample Application: UVa 469 - Wetlands of Florida

Let's see an example below (UVa 469 - Wetlands of Florida). The implicit graph is a 2D grid where the vertices are the cells in the grid and the edges are the connections between a cell and its S/SE/E/NE/N/NW/W/SW cells. 'W' denotes a wet cell and 'L' denotes a land cell. Wet area is defined as *connected cells* labeled with 'W'. We can label (and simultaneously count the size of) a wet area by using floodfill. The example below shows an execution of floodfill from row 2, column 1 (0-based indexing), replacing 'W' to '.'.

We want to make a remark that there are a good number of floodfill problems in UVa online judge [47] with a high profile example: UVa 1103 - Ancient Messages (ICPC World Finals problem in 2011). It may be beneficial for the readers to attempt floodfill problems listed in programming exercises of this section to master this technique!

```
// inside int main()
  // read the grid as a global 2D array + read (row, col) query coordinates
  printf("%d\n", floodfill(row, col, 'W', '.'));  // count size of wet area
                                                  // the returned answer is 12
// LLLLLLLLL         LLLLLLLLL
// LLWWLLWLL         LL..LLWLL   //         The size of connected component
// LWWLLLLLL (R2,C1) L..LLLLLL   //                     (the connected 'W's)
// LWWWLWWLL         L...L..LL   // with one 'W' at (row 2, column 1) is 12
// LLLWWWLLL ======> LLL...LLL
// LLLLLLLLL         LLLLLLLLL   //    Notice that all these connected 'W's
// LLLWWLLWL         LLLWWLLWL   //  are replaced with '.'s after floodfill
// LLWLWLLLL         LLWLWLLLL
// LLLLLLLLL         LLLLLLLLL
```

4.2.5 Topological Sort (Directed Acyclic Graph)

Topological sort (or topological ordering) of a Directed Acyclic Graph (DAG) is a linear ordering of the vertices in the DAG so that vertex u comes before vertex v if edge $(u \rightarrow v)$ exists in the DAG. Every DAG has at least one *and possibly more* topological sort(s).

One application of topological sorting is to find a possible sequence of modules that a University student has to take to fulfill graduation requirement. Each module has certain pre-requisites to be met. These pre-requisites are never cyclic, so they can be modeled as a DAG. Topological sorting this module pre-requisites DAG gives the student a linear list of modules to be taken one after another without violating the pre-requisites constraints.

There are several algorithms for topological sort. The simplest way is to slightly modify the DFS implementation we presented earlier in Section 4.2.1.

```
vi ts;              // global vector to store the toposort in reverse order

void dfs2(int u) {  // different function name compared to the original dfs
  dfs_num[u] = VISITED;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      dfs2(v.first);
  }
  ts.push_back(u); }                  // that's it, this is the only change
```

```
// inside int main()
  ts.clear();
  memset(dfs_num, UNVISITED, sizeof dfs_num);
  for (int i = 0; i < V; i++)       // this part is the same as finding CCs
    if (dfs_num[i] == UNVISITED)
      dfs2(i);
                // alternative, call: reverse(ts.begin(), ts.end()); first
  for (int i = (int)ts.size() - 1; i >= 0; i--)           // read backwards
    printf(" %d", ts[i]);
  printf("\n");

// For the sample graph in Figure 4.4, the output is like this:
// 7 6 0 1 2 5 3 4   (remember that there can be >= 1 valid toposort)
```

In `dfs2(u)`, we append u to the back of a list (vector) of explored vertices only after visiting all the subtrees below u in the DFS spanning tree[2]. We append u to the *back* of this vector because C++ STL `vector` (Java `Vector`) only supports *efficient* $O(1)$ *insertion* from the back. The list will be in reversed order, but we can work around this issue by reversing the print order in the output phase. This simple algorithm for finding (a valid) topological sort is due to Robert Endre Tarjan. It runs in $O(V+E)$ as with DFS as it does the same work as the original DFS plus one constant operation.

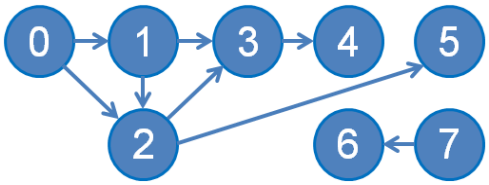


Figure 4.4: An Example of DAG

To complete the discussion about topological sort, we show another algorithm for finding topological sort: Kahn's algorithm [36]. It looks like a 'modified BFS'. Some problems, e.g. UVa 11060 - Beverages, requires this Kahn's algorithm to produce the required topological sort instead of the DFS-based algorithm shown earlier.

```
enqueue vertices with zero incoming degree into a (priority) queue Q;
while (Q is not empty) {
  vertex u = Q.dequeue(); put vertex u into a topological sort list;
  remove this vertex u and all outgoing edges from this vertex;
  if such removal causes vertex v to have zero incoming degree
    Q.enqueue(v); }
```

Exercise 4.2.5.1: Why appending vertex `u` at the back of `vi ts`, i.e. `ts.push_back(u)` in the standard DFS code is enough to help us find the topological sort of a DAG?

Exercise 4.2.5.2: Can you identify another data structure that supports efficient $O(1)$ insertion *from front* so that we do not have to reverse the content of `vi ts`?

Exercise 4.2.5.3: What happen if we run topological sort code above on a non DAG?

Exercise 4.2.5.4: The topological sort code shown above can only generate *one* valid topological ordering of the vertices of a DAG. What should we do if we want to output *all* valid topological orderings of the vertices of a DAG?

[2]DFS spanning tree is discussed in more details in Section 4.2.7.

4.2.6 Bipartite Graph Check

Bipartite graph has important applications that we will see later in Section 4.7.4. In this subsection, we just want to check if a graph is bipartite (or 2/bi-colorable) to solve problems like UVa 10004 - Bicoloring. We can use either BFS or DFS for this check, but we feel that BFS is more natural. The modified BFS code below starts by coloring the source vertex (first layer) with value 0, color the direct neighbors of the source vertex (second layer) with value 1, color the neighbors of direct neighbors (third layer) with value 0 again, and so on, alternating between value 0 and value 1 as the only two valid colors. If we encounter any violation(s) along the way—an edge with two endpoints having the same color, then we can conclude that the given input graph is not a bipartite graph.

```
// inside int main()
  queue<int> q; q.push(s);
  vi color(V, INF); color[s] = 0;
  bool isBipartite = true;  // addition of one boolean flag, initially true
  while (!q.empty() & isBipartite) { // similar to the original BFS routine
    int u = q.front(); q.pop();
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
      if (color[v.first] == INF) {   // but, instead of recording distance,
        color[v.first] = 1 - color[u];  // we just record two colors {0, 1}
        q.push(v.first); }
      else if (color[v.first] == color[u]) {  // u & v.first has same color
        isBipartite = false; break; } } }    // we have a coloring conflict
```

Exercise 4.2.6.1*: Implement bipartite check using DFS instead!

Exercise 4.2.6.2*: A *simple* graph with V vertices is found out to be a bipartite graph. What is the maximum possible number of edges that this graph has?

Exercise 4.2.6.3: Prove (or disprove) this statement: "Bipartite graph has no odd cycle"!

4.2.7 Graph Edges Property Check via DFS Spanning Tree

Running DFS on a connected graph generates a DFS *spanning tree*[3] (or *spanning forest*[4] if the graph is disconnected). With the help of one more vertex state: `EXPLORED = 2` (visited *but not yet completed*) on top of `VISITED` (visited *and completed*), we can use this DFS spanning tree (or forest) to classify graph edges into three types:

1. Tree edge: The edge traversed by DFS, i.e. an edge from a vertex currently with state: `EXPLORED` to a vertex with state: `UNVISITED`.
2. Back edge: Edge that is part of a cycle, i.e. an edge from a vertex currently with state: `EXPLORED` to a vertex with state: `EXPLORED` too. This is an important application of this algorithm. Note that we usually do not count bi-directional edges as having a 'cycle' (We need to remember `dfs_parent` to distinguish this, see the code below).
3. Forward/Cross edges from vertex with state: `EXPLORED` to vertex with state: `VISITED`. These two type of edges are not typically tested in programming contest problems.

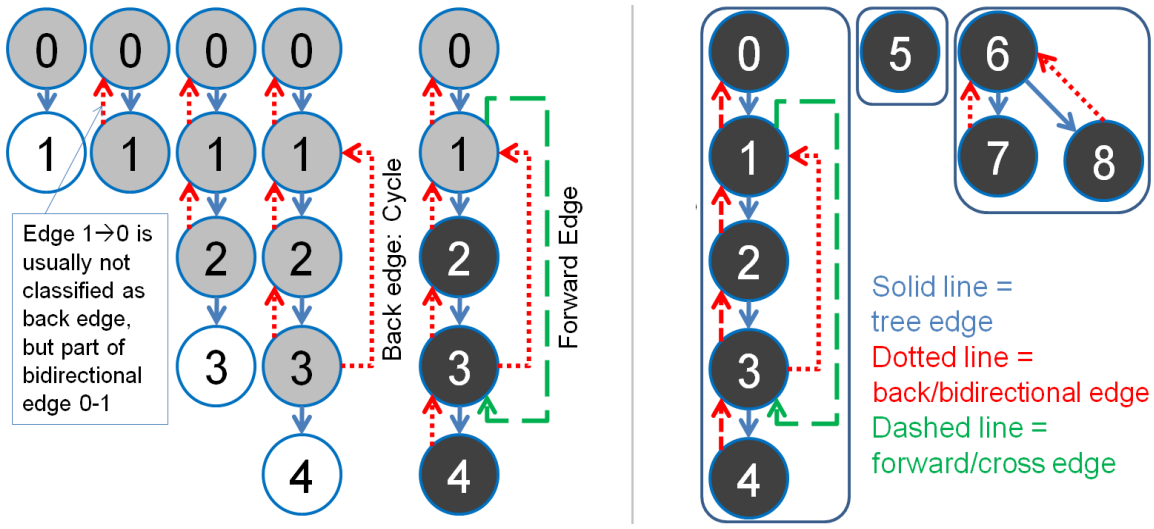


Figure 4.5: Animation of DFS when Run on the Sample Graph in Figure 4.1

Figure 4.5 shows an animation (from left to right) of calling `dfs(0)` (shown in more details), then `dfs(5)`, and finally `dfs(6)` on the sample graph in Figure 4.1. We can see that $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ is a (true) cycle and we classify edge $(3 \rightarrow 1)$ as a back edge, whereas $0 \rightarrow 1 \rightarrow 0$ is not a cycle but it is just a bi-directional edge (0-1). The code for this DFS variant is shown below.

```
void graphCheck(int u) {          // DFS for checking graph edge properties
  dfs_num[u] = EXPLORED;          // color u as EXPLORED instead of VISITED
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED) {  // Tree Edge, EXPLORED->UNVISITED
      dfs_parent[v.first] = u;            // parent of this children is me
      graphCheck(v.first);
    }
    else if (dfs_num[v.first] == EXPLORED) {          // EXPLORED->EXPLORED
      if (v.first == dfs_parent[u])     // to differentiate these two cases
        printf(" Two ways (%d, %d)-(%d, %d)\n", u, v.first, v.first, u);
      else   // the most frequent application: check if the graph is cyclic
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
    }
    else if (dfs_num[v.first] == VISITED)              // EXPLORED->VISITED
      printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
  }
  dfs_num[u] = VISITED;       // after recursion, color u as VISITED (DONE)
}

// inside int main()
  dfs_num.assign(V, UNVISITED);
  dfs_parent.assign(V, 0);                                    // new vector
```

[3]A spanning tree of a connected graph G is a tree that spans (covers) all vertices of G but only using a subset of the edges of G.

[4]A disconnected graph G has several connected components. Each component has its own spanning subtree(s). All spanning subtrees of G, one from each component, form what we call a spanning forest.

```
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED)
      printf("Component %d:\n", ++numComp), graphCheck(i); // 2 lines in 1!

// For the sample graph in Figure 4.1, the output is like this:
// Component 1:
//  Two ways (1, 0) - (0, 1)
//  Two ways (2, 1) - (1, 2)
//  Back Edge (3, 1) (Cycle)
//  Two ways (3, 2) - (2, 3)
//  Two ways (4, 3) - (3, 4)
//  Forward/Cross Edge (1, 3)
// Component 2:
// Component 3:
//  Two ways (7, 6) - (6, 7)
//  Two ways (8, 6) - (6, 8)
```

Exercise 4.2.7.1: Perform graph edges property check on the graph in Figure 4.9. Assume that you start DFS from vertex 0. How many back edges that you can find this time?

4.2.8 Finding Articulation Points and Bridges (Undirected Graph)

Motivating problem: Given a road map (undirected graph) with sabotage costs associated to all intersections (vertices) and roads (edges), sabotage either a single intersection or a single road such that the road network breaks down (disconnected) and do so in the least cost way. This is a problem of finding the least cost Articulation Point (intersection) or the least cost Bridge (road) in an undirected graph (road map).

An 'Articulation Point' is defined as *a vertex* in a graph G whose removal (all edges incident to this vertex are also removed) disconnects G. A graph without any articulation point is called 'Biconnected'. Similarly, a 'Bridge' is defined as *an edge* in a graph G whose removal disconnects G. These two problems are usually defined for undirected graphs (they are more challenging for directed graphs and require another algorithm to solve, see [35]).

A naïve algorithm to find articulation points is as follows (can be tweaked to find bridges):

1. Run $O(V+E)$ DFS (or BFS) to count number of connected components (CCs) of the original graph. Usually, the input is a connected graph, so this check will usually gives us one connected component.
2. For each vertex $v \in V$ // $O(V)$
   (a) Cut (remove) vertex v and its incident edges
   (b) Run $O(V+E)$ DFS (or BFS) and see if the number of CCs increases
   (c) If yes, v is an articulation point/cut vertex; Restore v and its incident edges

This naïve algorithm calls DFS (or BFS) $O(V)$ times, thus it runs in $O(V \times (V+E)) = O(V^2 + VE)$. But this is *not* the best algorithm as we can actually just run the $O(V+E)$ DFS *once* to identify all the articulation points and bridges.

This DFS variant, due to John Edward Hopcroft and Robert Endre Tarjan (see [63] and problem 22.2 in [7]), is just another extension from the previous DFS code shown earlier.

We now maintain two numbers: `dfs_num(u)` and `dfs_low(u)`. Here, `dfs_num(u)` now stores the iteration counter when the vertex u is visited *for the first time* (not just for distinguishing `UNVISITED` versus `EXPLORED`/`VISITED`). The other number `dfs_low(u)` stores the lowest `dfs_num` reachable from the current DFS spanning subtree of u. At the beginning, `dfs_low(u) = dfs_num(u)` when vertex u is visited for the first time. Then, `dfs_low(u)` can only be made smaller if there is a cycle (a back edge exists). Note that we do not update `dfs_low(u)` with a back edge (u, v) if v is a direct parent of u.

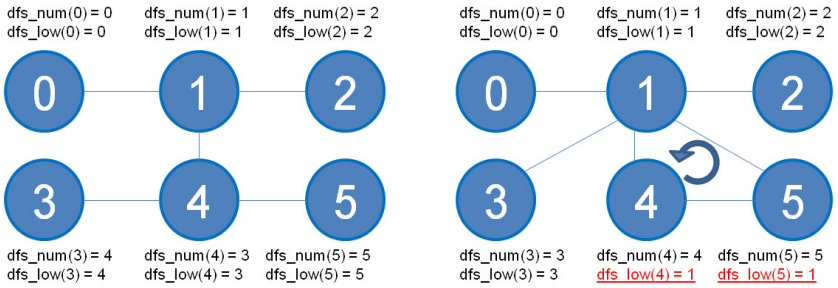


Figure 4.6: Introducing two More DFS Attributes: `dfs_num` and `dfs_low`

See Figure 4.6 for clarity. In both graphs, we run the DFS variant from vertex 0. Suppose for the graph in Figure 4.6—left side, the sequence of visitation is 0 (at iteration 0) → 1 (1) → 2 (2) (backtrack to 1) → 4 (3) → 3 (4) (backtrack to 4) → 5 (5). See that these iteration counters are shown correctly in `dfs_num`. As there is no back edge in this graph, all `dfs_low = dfs_num`.

Suppose for the graph in Figure 4.6—right side, the sequence of visitation is 0 (at iteration 0) → 1 (1) → 2 (2) (backtrack to 1) → 3 (3) (backtrack to 1) → 4 (4) → 5 (5). At this point in the DFS spanning tree, there is an important back edge that forms a cycle, i.e. edge 5-1 that is part of cycle 1-4-5-1. This causes vertices 1, 4, and 5 to be able to reach vertex 1 (with `dfs_num` 1). Thus `dfs_low` of {1, 4, 5} are all 1.

When we are in a vertex u with v as its neighbor and `dfs_low(v)` $\geq$ `dfs_num(u)`, then u is an articulation vertex. This is because the fact that `dfs_low(v)` is *not smaller* than `dfs_num(u)` implies that there is *no back edge* from vertex v that can reach another vertex w with a lower `dfs_num(w)` than `dfs_num(u)`. A vertex w with lower `dfs_num(w)` than vertex u with `dfs_num(u)` implies that w is the ancestor of u in the DFS spanning tree. This means that to reach the ancestor(s) of u from v, one *must* pass through vertex u. Therefore, removing vertex u will disconnect the graph.

However, there is one **special case**: The root of the DFS spanning tree (the vertex chosen as the start of DFS call) is an articulation point only if it has more than one children in the DFS spanning tree (a trivial case that is not detected by this algorithm).

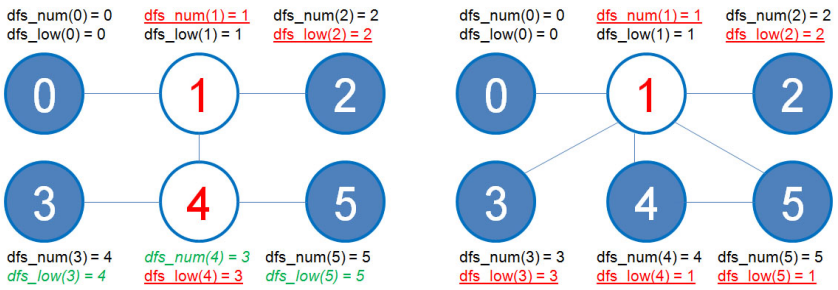


Figure 4.7: Finding Articulation Points with `dfs_num` and `dfs_low`

See Figure 4.7 for more details. On the graph in Figure 4.7—left side, vertices 1 and 4 are articulation points, because for example in edge 1-2, we see that `dfs_low(2)` $\geq$ `dfs_num(1)`

and in edge 4-5, we also see that `dfs_low(5)` $\geq$ `dfs_num(4)`. On the graph in Figure 4.7—right side, only vertex 1 is the articulation point, because for example in edge 1-5, `dfs_low(5)` $\geq$ `dfs_num(1)`.

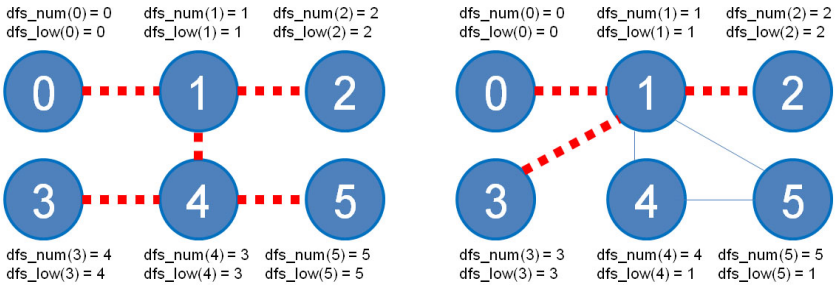


Figure 4.8: Finding Bridges, also with `dfs_num` and `dfs_low`

The process to find bridges is similar. When `dfs_low(v)` > `dfs_num(u)`, then edge u-v is a bridge (notice that we remove the equality test '=' for finding bridges). In Figure 4.8, almost all edges are bridges for the left and right graph. Only edges 1-4, 4-5, and 5-1 are not bridges on the right graph (they actually form a cycle). This is because—for example—for edge 4-5, we have `dfs_low(5)` $\leq$ `dfs_num(4)`, i.e. even if this edge 4-5 is removed, we know for sure that vertex 5 can still reach vertex 1 via *another path* that bypass vertex 4 as `dfs_low(5) = 1` (that other path is actually edge 5-1). The code is shown below:

```
void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED) {                    // a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;      // special case if u is a root

      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])         // for articulation point
        articulation_vertex[u] = true;      // store this information first
      if (dfs_low[v.first] > dfs_num[u])                      // for bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);  // update dfs_low[u]
    }
    else if (v.first != dfs_parent[u])  // a back edge and not direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);  // update dfs_low[u]
} }

// inside int main()
  dfsNumberCounter = 0; dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0);
  dfs_parent.assign(V, 0); articulation_vertex.assign(V, 0);
  printf("Bridges:\n");
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED) {
      dfsRoot = i; rootChildren = 0; articulationPointAndBridge(i);
      articulation_vertex[dfsRoot] = (rootChildren > 1); }  // special case
```

```
printf("Articulation Points:\n");
for (int i = 0; i < V; i++)
  if (articulation_vertex[i])
    printf(" Vertex %d\n", i);
```

Exercise 4.2.8.1: Examine the graph in Figure 4.1 without running the algorithm above. Which vertices are articulation points and which edges are bridges? Now run the algorithm and verify if the computed `dfs_num` and `dfs_low` of each vertex of Figure 4.1 graph can be used to identify the same articulation points and bridges found manually!

4.2.9 Finding Strongly Connected Components (Directed Graph)

Yet another application of DFS is to find *strongly* connected components in a *directed* graph, e.g. UVa 11838 - Come and Go. This is a different problem to finding connected components in an undirected graph. In Figure 4.9, we have a similar graph to the graph in Figure 4.1, but now the edges are directed. Although the graph in Figure 4.9 looks like it has one 'connected' component, it is actually not a 'strongly connected' component. In directed graphs, we are more interested with the notion of 'Strongly Connected Component (SCC)'. An SCC is defined as such: If we pick any pair of vertices u and v in the SCC, we can find a path from u to v and vice versa. There are actually three SCCs in Figure 4.9, as highlighted with the three boxes: {0}, {1, 3, 2}, and {4, 5, 7, 6}. Note: If these SCCs are contracted (replaced by larger vertices), they form a DAG (also see Section 8.4.3).

There are at least two known algorithms to find SCCs: Kosaraju's—explained in [7] and Tarjan's algorithm [63]. In this section, we adopt Tarjan's version, as it extends naturally from our previous discussion of finding Articulation Points and Bridges—also due to Tarjan. We will discuss Kosaraju's algorithm later in Section 9.17.

The basic idea of the algorithm is that SCCs form subtrees in the DFS spanning tree (compare the original directed graph and the DFS spanning tree in Figure 4.9). On top of computing `dfs_num(u)` and `dfs_low(u)` for each vertex, we also append vertex u to the back of a stack S (here the stack is implemented with a vector) and keep track of the vertices that are currently explored via `vi visited`. The condition to update `dfs_low(u)` is slightly different from the previous DFS algorithm for finding articulation points and bridges. Here, only vertices that currently have `visited` flag turned on (part of the current SCC) that can update `dfs_low(u)`. Now, if we have vertex u in this DFS spanning tree with `dfs_low(u) = dfs_num(u)`, we can conclude that u is the root (start) of an SCC (observe vertex 0, 1, and 4) in Figure 4.9) and the members of those SCCs are identified by popping the current content of stack S until we reach vertex u (the root) of SCC again.

In Figure 4.9, the content of S is $\{0, 1, 3, 2, \underline{4, 5, 7, 6}\}$ when vertex 4 is identified as the root of an SCC (`dfs_low(4) = dfs_num(4) = 4`), so we pop elements in S one by one until we reach vertex 4 and we have this SCC: {6, 7, 5, 4}. Next, the content of S is $\{0, \underline{1, 3, 2}\}$ when vertex 1 is identified as another root of another SCC (`dfs_low(1) = dfs_num(1) = 1`), so we pop elements in S one by one until we reach vertex 1 and we have SCC: {2, 3, 1}. Finally, we have the last SCC with one member only: {0}.

The code given below explores the directed graph and reports its SCCs. This code is basically a tweak of the standard DFS code. The recursive part is similar to standard DFS and the SCC reporting part will run in amortized $O(V)$ times, as each vertex will only belong to one SCC and thus reported only once. In overall, this algorithm still runs in $O(V + E)$.

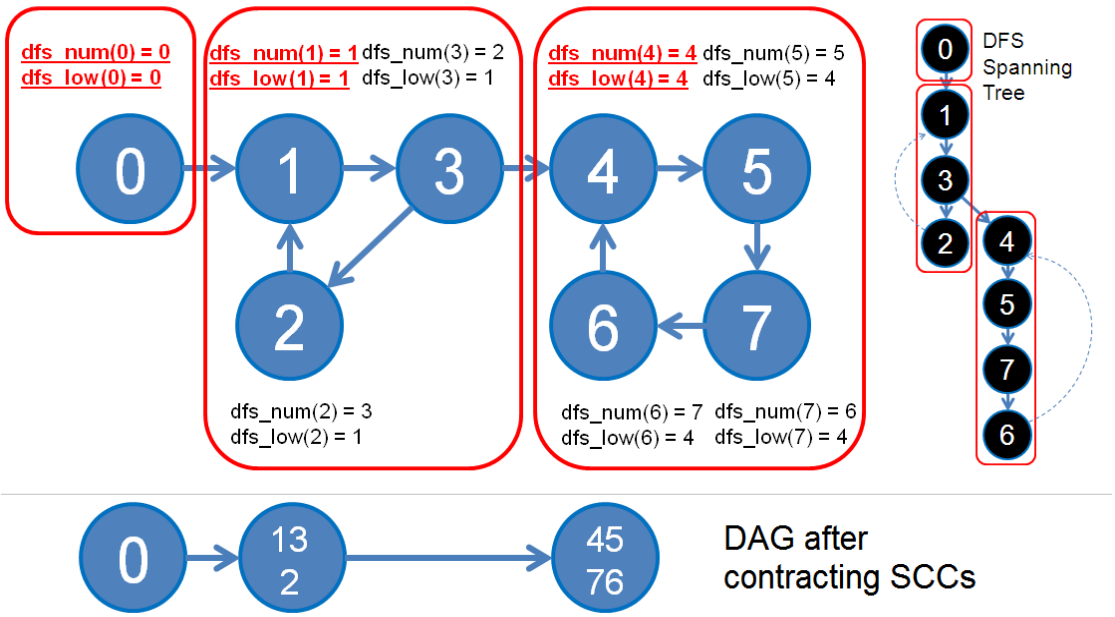


Figure 4.9: An Example of a Directed Graph and its SCCs

```
vi dfs_num, dfs_low, S, visited;                          // global variables

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++; // dfs_low[u] <= dfs_num[u]
  S.push_back(u);       // stores u in a vector based on order of visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == UNVISITED)
      tarjanSCC(v.first);
    if (visited[v.first])                              // condition for update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]); }

  if (dfs_low[u] == dfs_num[u]) {    // if this is a root (start) of an SCC
    printf("SCC %d:", ++numSCC);       // this part is done after recursion
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break; }
    printf("\n");
} }

// inside int main()
  dfs_num.assign(V, UNVISITED); dfs_low.assign(V, 0); visited.assign(V, 0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == UNVISITED)
      tarjanSCC(i);
```

Source code: ch4_01_dfs.cpp/java; ch4_02_UVa469.cpp/java

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!

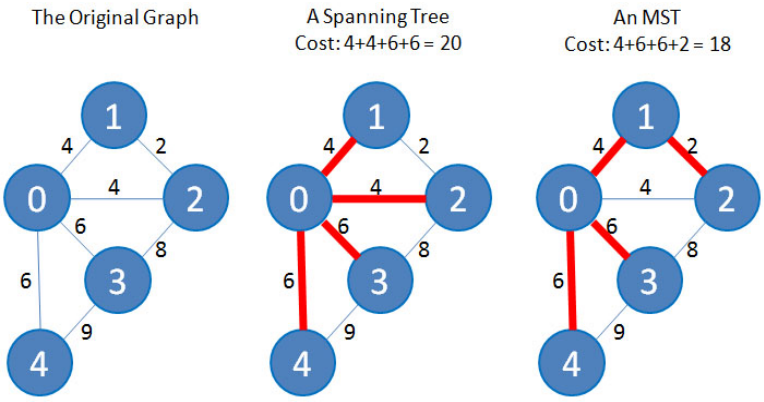


Figure 4.10: Example of an MST Problem

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.

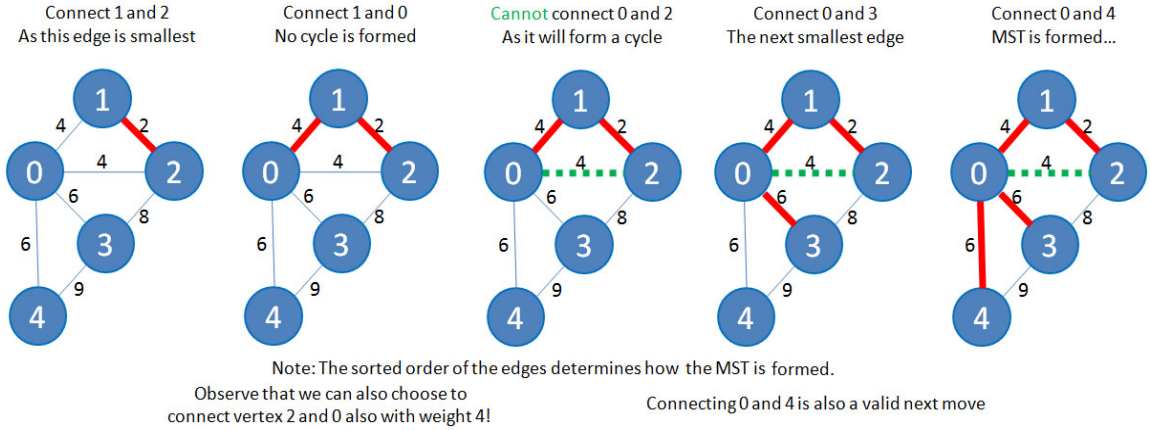


Figure 4.11: Animation of Kruskal's Algorithm for an MST Problem

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.

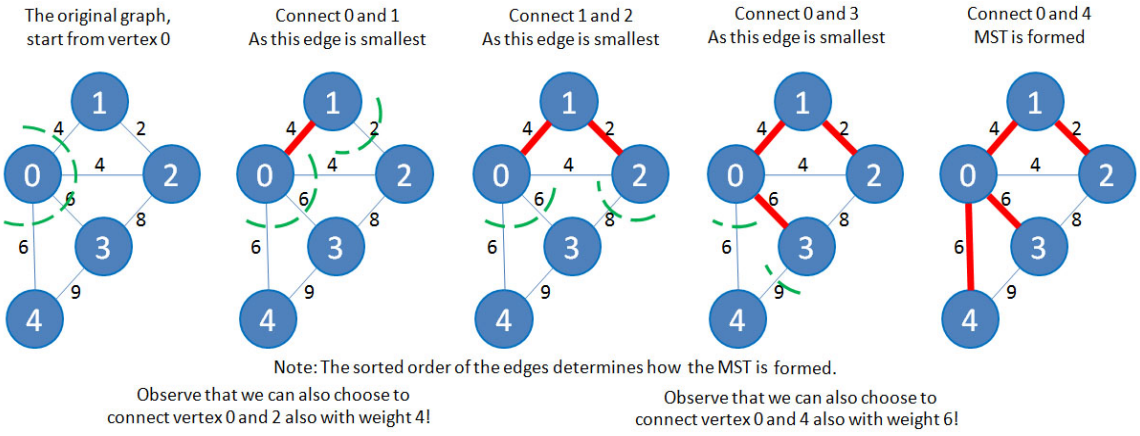


Figure 4.12: Animation of Prim's Algorithm for the same graph as in Figure 4.10—left

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.

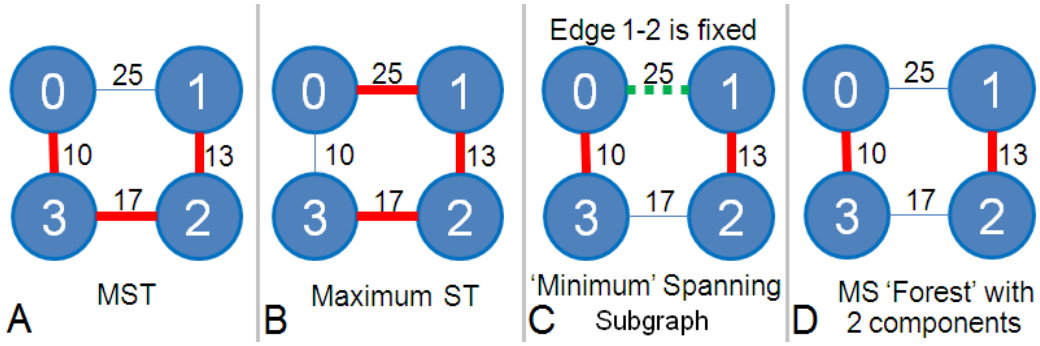


Figure 4.13: From left to right: MST, 'Maximum' ST, 'Minimum' SS, MS 'Forest'

'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

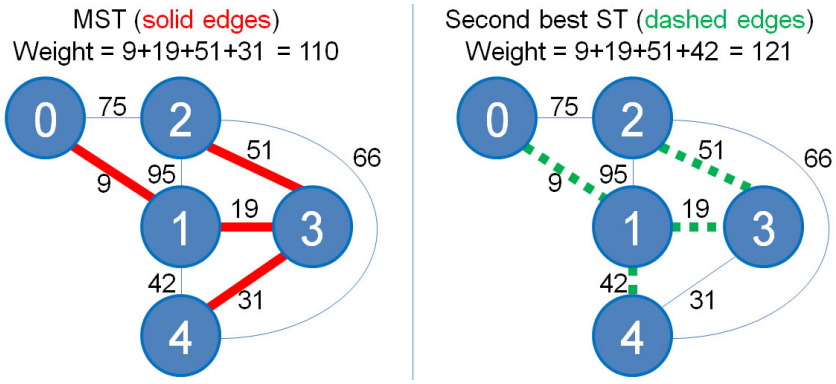


Figure 4.14: Second Best ST (from UVa 10600 [47])

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 $O(\text{sort the edges once} + \text{find the original MST} + \text{find the second best ST}) = O(E \log V + E + VE) = O(VE)$.

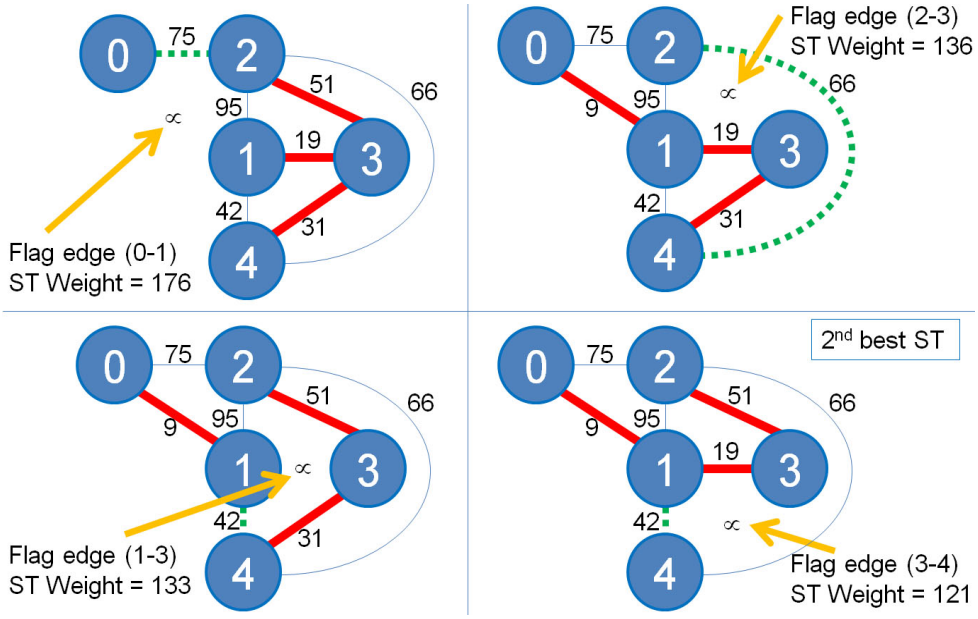


Figure 4.15: Finding the Second Best Spanning Tree from the MST

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

Minimax (and Maximin)

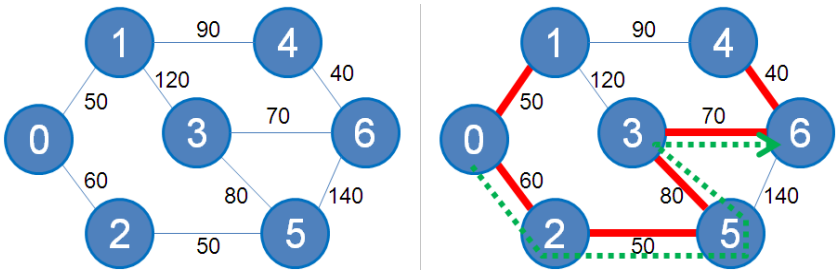


Figure 4.16: Minimax (UVa 10048 [47])

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.

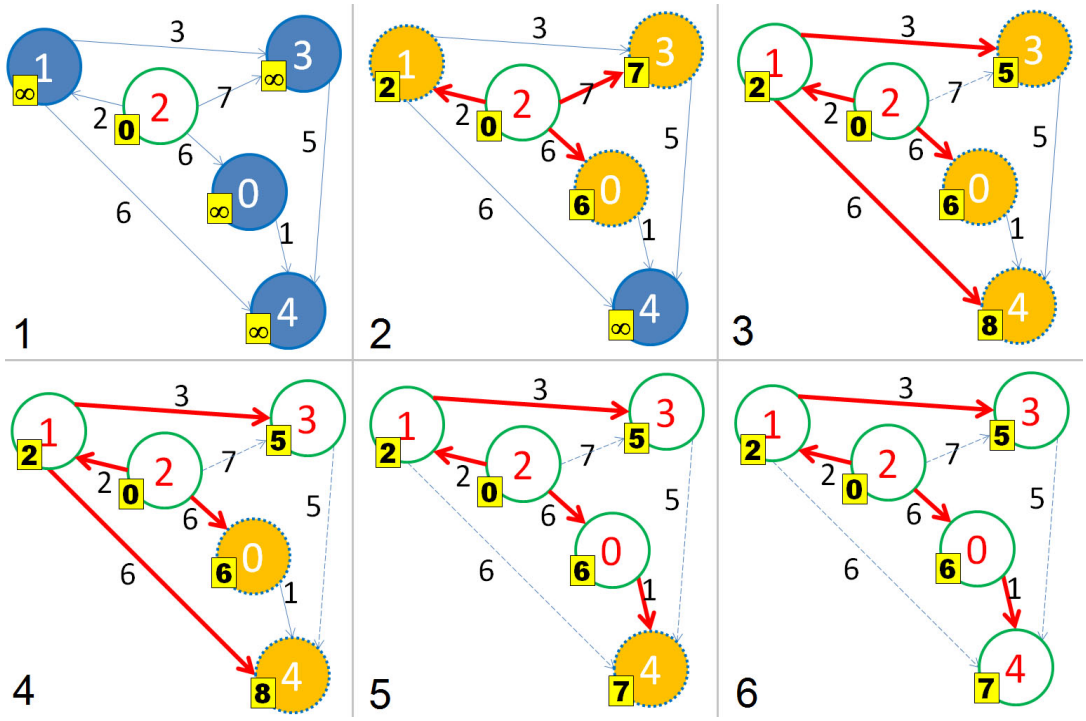


Figure 4.17: Dijkstra Animation on a Weighted Graph (from UVa 341 [47])

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.

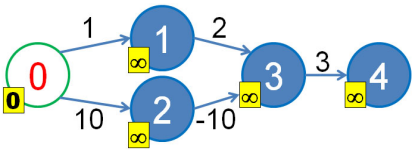


Figure 4.18: -ve Weight

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.

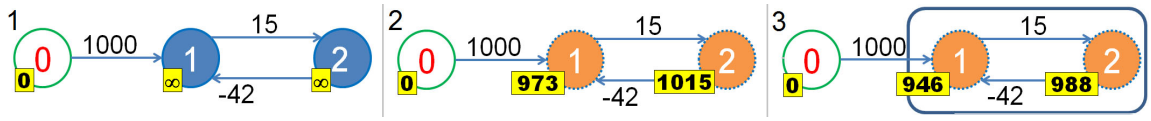


Figure 4.19: Bellman Ford's can detect the presence of negative cycle (from UVa 558 [47])

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)?

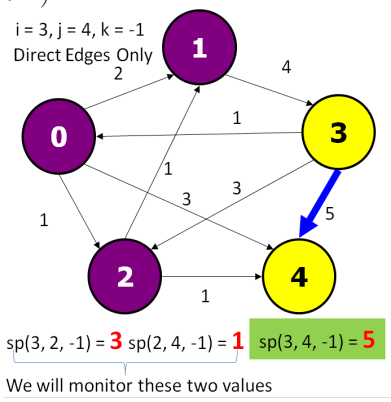
Exercise 4.5.1.2: In Section 4.4.4, we differentiate graph with negative weight edges but no negative cycle and graph with negative cycle. Will this short Floyd Warshall's algorithm works on graph with negative weight and/or negative cycle? Do some experiment!

---

[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).

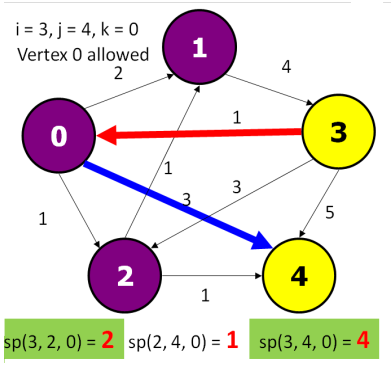


The current content of Adjacency Matrix D at **k = -1**

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.

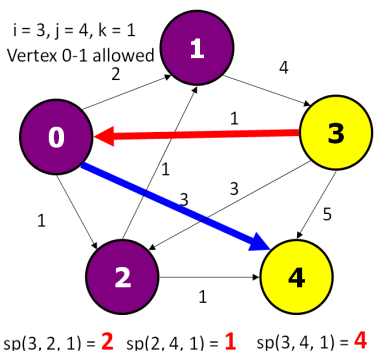


The current content of Adjacency Matrix D at **k = 0**

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.

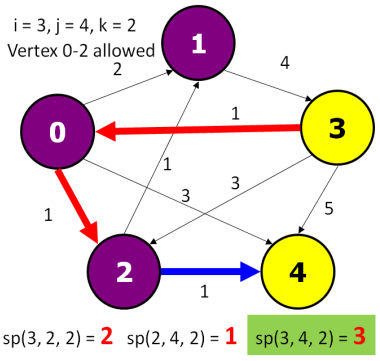


The current content of Adjacency Matrix D at **k = 1**

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 $O(V^3)$.

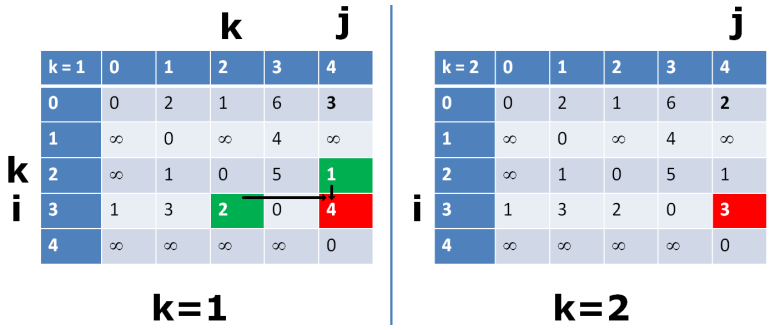


Table 4.3: Floyd Warshall's DP Table

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.

sub-problem is solvable with the four liner Floyd Warshall's (e.g. UVa 10171, 10793, 11463). A non competitive programmer will take longer route to deal with this sub-problem.

According to our experience, many shortest paths problems are not on weighted graphs that require Dijkstra's or Floyd Warshall's algorithms. If you look at the programming exercises listed in Section 4.4 (and later in Section 8.2), you will see that many of them are on unweighted graphs that are solvable with BFS (see Section 4.4.2).

We also observe that today's trend related to shortest paths problem involves careful *graph modeling* (UVa 10067, 10801, 11367, 11492, 12160). Therefore, to do well in programming contests, make sure that you have this soft skill: The ability to spot the graph in the problem statement. We have shown several examples of such graph modeling skill in this chapter which we hope you are able to appreciate and eventually make it yours.

In Section 4.7.1, we will revisit some shortest paths problems on Directed Acyclic Graph (DAG). This important variant is solvable with generic Dynamic Programming (DP) technique that have been discussed in Section 3.5. We will also present another way of viewing DP technique as 'algorithm on DAG' in that section.

We present an SSSP/APSP algorithm decision table within the context of programming contest in Table 4.4 to help the readers in deciding which algorithm to choose depending on various graph criteria. The terminologies used are as follows: 'Best' → the most suitable algorithm; 'Ok' → a correct algorithm but not the best; 'Bad' → a (very) slow algorithm; 'WA' → an incorrect algorithm; and 'Overkill' → a correct algorithm but unnecessary.

Graph Criteria	BFS $O(V+E)$	Dijkstra's $O((V+E)\log V)$	Bellman Ford's $O(VE)$	Floyd Warshall's $O(V^3)$
Max Size	$V, E \leq 10M$	$V, E \leq 300K$	$VE \leq 10M$	$V \leq 400$
Unweighted	Best	Ok	Bad	Bad in general
Weighted	WA	Best	Ok	Bad in general
Negative weight	WA	Our variant Ok	Ok	Bad in general
Negative cycle	Cannot detect	Cannot detect	Can detect	Can detect
Small graph	WA if weighted	Overkill	Overkill	Best

Table 4.4: SSSP/APSP Algorithm Decision Table

Programming Exercises for Floyd Warshall's algorithm:

- Floyd Warshall's Standard Application (for APSP or SSSP on small graph)
  1. UVa 00341 - Non-Stop Travel (graph is small)
  2. UVa 00423 - MPI Maelstrom (graph is small)
  3. UVa 00567 - Risk (simple SSSP solvable with BFS, but graph is small, so can be solved easier with Floyd Warshall's)
  4. **UVa 00821 - Page Hopping *** (LA 5221, World Finals Orlando00, one of the 'easiest' ICPC World Finals problem)
  5. UVa 01233 - USHER (LA 4109, Singapore07, Floyd Warshall's can be used to find the minimum cost cycle in the graph; the maximum input graph size is $p \leq 500$ but still doable in UVa online judge)
  6. UVa 01247 - Interstar Transport (LA 4524, Hsinchu09, APSP, Floyd Warshall's, modified a bit to prefer shortest path with less intermediate vertices)
  7. **UVa 10171 - Meeting Prof. Miguel *** (easy with APSP information)
  8. *UVa 10354 - Avoiding Your Boss* (find boss's shortest paths, remove edges involved in boss's shortest paths, re-run shortest paths from home to market)

9. *UVa 10525 - New to Bangladesh?* (use two adjacency matrices: time and length; use modified Floyd Warshall's)
10. UVa 10724 - Road Construction (adding one edge only changes 'a few things')
11. UVa 10793 - The Orc Attack (Floyd Warshall's simplifies this problem)
12. UVa 10803 - Thunder Mountain (graph is small)
13. UVa 10947 - Bear with me, again.. (graph is small)
14. UVa 11015 - 05-32 Rendezvous (graph is small)
15. **UVa 11463 - Commandos *** (solution is easy with APSP information)
16. *UVa 12319 - Edgetown's Traffic Jams* (Floyd Warshall's 2x and compare)

- Variants
  1. **UVa 00104 - Arbitrage *** (small arbitrage problem solvable with FW)
  2. UVa 00125 - Numbering Paths (modified Floyd Warshall's)
  3. UVa 00186 - Trip Routing (graph is small, print path)
  4. *UVa 00274 - Cat and Mouse* (variant of transitive closure problem)
  5. UVa 00436 - Arbitrage (II) (another arbitrage problem)
  6. **UVa 00334 - Identifying Concurrent ... *** (transitive closure++)
  7. UVa 00869 - Airline Comparison (run Warshall's 2x, compare AdjMatrices)
  8. *UVa 00925 - No more prerequisites ...* (transitive closure++)
  9. ***UVa 01056 - Degrees of Separation **** (LA 3569, World Finals SanAntonio06, diameter of a small graph)
  10. *UVa 01198 - Geodetic Set Problem* (LA 2818, Kaohsiung03, trans closure++)
  11. UVa 11047 - The Scrooge Co Problem (print path; special case: if origin = destination, print twice)

Profile of Algorithm Inventors

Robert W Floyd (1936-2001) was an eminent American computer scientist. Floyd's contributions include the design of **Floyd's algorithm** [19], which efficiently finds all shortest paths in a graph. Floyd worked closely with Donald Ervin Knuth, in particular as the major reviewer for Knuth's seminal book The Art of Computer Programming, and is the person most cited in that work.

Stephen Warshall (1935-2006) was a computer scientist who invented the **transitive closure algorithm**, now known as **Warshall's algorithm** [70]. This algorithm was later named as Floyd Warshall's as Floyd independently invented essentially similar algorithm.

Jack R. Edmonds (born 1934) is a mathematician. He and Richard Karp invented the **Edmonds Karp's algorithm** for computing the Max Flow in a flow network in $O(VE^2)$ [14]. He also invented an algorithm for MST on directed graphs (Arborescence problem). This algorithm was proposed independently first by Chu and Liu (1965) and then by Edmonds (1967)—thus called the **Chu Liu/Edmonds's algorithm** [6]. However, his most important contribution is probably the Edmonds's **matching/blossom shrinking algorithm**—one of the most cited Computer Science papers [13].

Richard Manning Karp (born 1935) is a computer scientist. He has made many important discoveries in computer science in the area of combinatorial algorithms. In 1971, he and Edmonds published the **Edmonds Karp's algorithm** for solving the Max Flow problem [14]. In 1973, he and John Hopcroft published the **Hopcroft Karp's algorithm**, still the fastest known method for finding Maximum Cardinality Bipartite Matching [28].

4.6 Network Flow

4.6.1 Overview and Motivation

Motivating problem: Imagine a connected, (integer) weighted, and directed graph[13] as a pipe network where the edges are the pipes and the vertices are the splitting points. Each edge has a weight equals to the capacity of the pipe. There are also two special vertices: source s and sink t. What is the maximum flow (rate) from source s to sink t in this graph (imagine water flowing in the pipe network, we want to know the maximum volume of water over time that can pass through this pipe network)? This problem is called the Maximum Flow problem (often abbreviated as just Max Flow), one of the problems in the family of problems involving flow in networks. See an illustration of Max Flow in Figure 4.24.

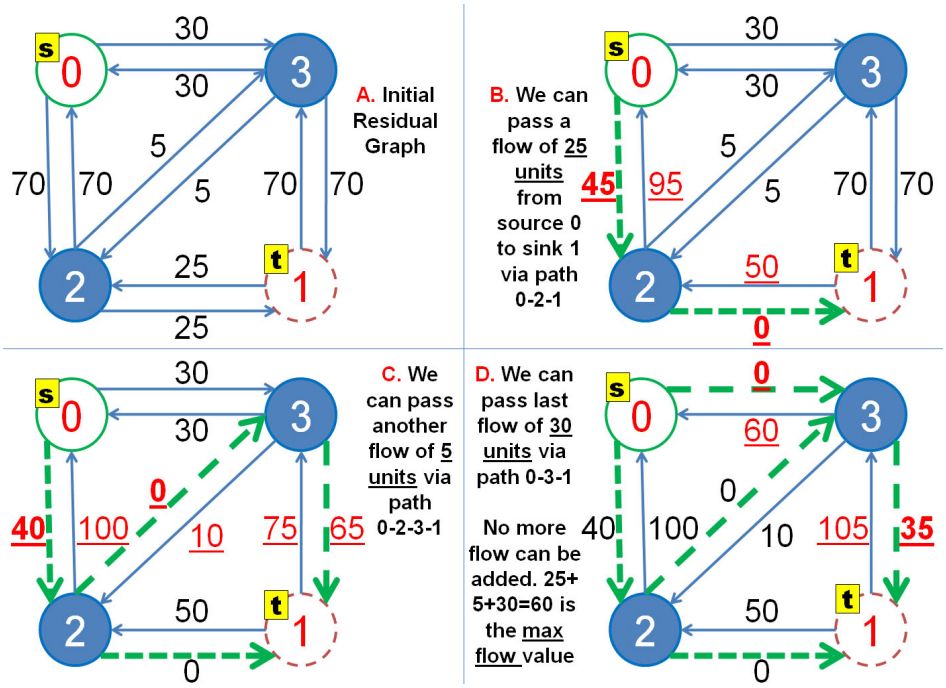


Figure 4.24: Max Flow Illustration (UVa 820 [47] - ICPC World Finals 2000 Problem E)

4.6.2 Ford Fulkerson's Method

One solution for Max Flow is the Ford Fulkerson's method—invented by the same Lester Randolph *Ford*, Jr who invented the Bellman Ford's algorithm and Delbert Ray *Fulkerson*.

```
setup directed residual graph with edge capacity = original graph weights
mf = 0            // this is an iterative algorithm, mf stands for max_flow
while (there exists an augmenting path p from s to t) {
  // p is a path from s to t that pass through +ve edges in residual graph
  augment/send flow f along the path p (s -> ... -> i -> j -> ... t)
    1. find f, the minimum edge weight along the path p
    2. decrease capacity of forward edges (e.g. i -> j) along path p by f
    3. increase capacity of backward edges (e.g. j -> i) along path p by f
  mf += f          // we can send a flow of size f from s to t, increase mf
}
output mf                                        // this is the max flow value
```

[13] A weighted edge in an undirected graph can be transformed to two directed edges with the same weight.

Ford Fulkerson's method is an iterative algorithm that repeatedly finds augmenting path p: A path from source s to sink t that passes through positive weighted edges in the residual[14] graph. After finding an augmenting path p that has f as the minimum edge weight along the path p (the bottleneck edge in this path), Ford Fulkerson's method will do two important steps: Decreasing/increasing the capacity of forward $(i \rightarrow j)$/backward $(j \rightarrow i)$ edges along path p by f, respectively. Ford Fulkerson's method will repeat this process until there is no more possible augmenting path from source s to sink t anymore which implies that the total flow so far is the maximum flow. Now see Figure 4.24 again with this understanding.

The reason for decreasing the capacity of forward edge is obvious. By sending a flow through augmenting path p, we will decrease the remaining (residual) capacities of the (forward) edges used in p. The reason for increasing the capacity of backward edges may not be that obvious, but this step is important for the correctness of Ford Fulkerson's method. By increasing the capacity of a backward edge $(j \rightarrow i)$, Ford Fulkerson's method allows *future iteration (flow)* to cancel (part of) the capacity used by a forward edge $(i \rightarrow j)$ that was incorrectly used by some earlier flow(s).

There are several ways to find an augmenting s-t path in the pseudo code above, each with different behavior. In this section, we highlight two ways: via DFS or via BFS.

Ford Fulkerson's method implemented using DFS may run in $O(|f^*|E)$ where $|f^*|$ is the Max Flow `mf` value. This is because we can have a graph like in Figure 4.25. Then, we may encounter a situation where two augmenting paths: $s \rightarrow a \rightarrow b \rightarrow t$ and $s \rightarrow b \rightarrow a \rightarrow t$ only decrease the (forward[15]) edge capacities along the path by 1. In the worst case, this is repeated $|f^*|$ times (it is 200 times in Figure 4.25). Because DFS runs in $O(E)$ in a flow graph[16], the overall time complexity is $O(|f^*|E)$. We do not want this unpredictability in programming contests as the problem author can choose to give a (very) large $|f^*|$ value.

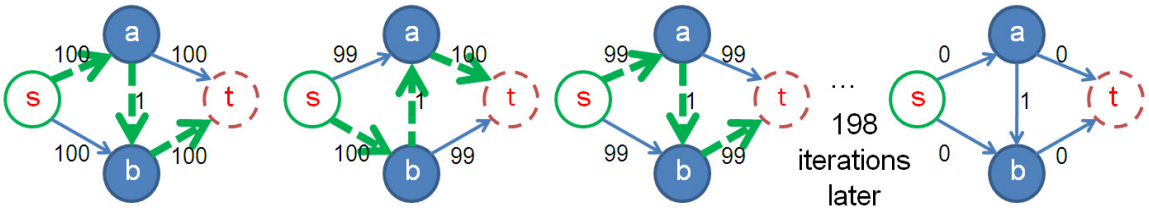


Figure 4.25: Ford Fulkerson's Method Implemented with DFS Can Be Slow

4.6.3 Edmonds Karp's Algorithm

A better implementation of the Ford Fulkerson's method is to use BFS for finding the shortest path in terms of number of layers/hops between s and t. This algorithm was discovered by Jack *Edmonds* and Richard Manning *Karp*, thus named as Edmonds Karp's algorithm [14]. It runs in $O(VE^2)$ as it can be proven that after $O(VE)$ BFS iterations, all augmenting paths will already be exhausted. Interested readers can browse references like [14, 7] to study more about this proof. As BFS runs in $O(E)$ in a flow graph, the overall time complexity is $O(VE^2)$. Edmonds Karp's only needs two s-t paths in Figure 4.25: s→a→t (2 hops, send

[14]We use the name 'residual graph' because initially the weight of each edge `res[i][j]` is the same as the original capacity of edge `(i, j)` in the original graph. If this edge `(i, j)` is used by an augmenting path and a flow pass through this edge with weight `f` $\leq$ `res[i][j]` (a flow cannot exceed this capacity), then the remaining (or residual) capacity of edge `(i, j)` will be `res[i][j] - f`.

[15]Note that after sending flow $s \rightarrow a \rightarrow b \rightarrow t$, the forward edge $a \rightarrow b$ is replaced by the backward edge $b \rightarrow a$, and so on. If this is not so, then the max flow value is just $1 + 99 + 99 = 199$ instead of 200 (wrong).

[16]The number of edges in a flow graph must be $E \geq V - 1$ to ensure $\exists \geq 1$ s-t flow. This implies that both DFS and BFS—using Adjacency List—run in $O(E)$ instead of $O(V + E)$.

100 unit flow) and s→b→t (2 hops, send another 100). That is, it does not get trapped to send flow via the longer paths (3 hops): s→a→b→t (or s→b→a→t).

Coding the Edmonds Karp's algorithm for the first time can be a challenge for new programmers. In this section, we provide our simplest Edmonds Karp's code that uses *only* Adjacency Matrix named as `res` with size $O(V^2)$ to store the residual capacity of each edge. This version—which runs in $O(VE)$ BFS iterations $\times$ $O(V^2)$ per BFS due to Adjacency Matrix = $O(V^3E)$—is fast enough to solve *some* (small-size) Max Flow problems.

```
int res[MAX_V][MAX_V], mf, f, s, t;                    // global variables
vi p;                                 // p stores the BFS spanning tree from s

void augment(int v, int minEdge) {  // traverse BFS spanning tree from s->t
  if (v == s) { f = minEdge; return; }  // record minEdge in a global var f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v]));
                         res[p[v]][v] -= f; res[v][p[v]] += f; } }

// inside int main(): set up 'res', 's', and 't' with appropriate values
  mf = 0;                                              // mf stands for max_flow
  while (1) {        // O(VE^2) (actually O(V^3 E) Edmonds Karp's algorithm
    f = 0;
    // run BFS, compare with the original BFS shown in Section 4.2.2
    vi dist(MAX_V, INF); dist[s] = 0; queue<int> q; q.push(s);
    p.assign(MAX_V, -1);      // record the BFS spanning tree, from s to t!
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break; // immediately stop BFS if we already reach sink t
      for (int v = 0; v < MAX_V; v++)            // note: this part is slow
        if (res[u][v] > 0 && dist[v] == INF)
          dist[v] = dist[u] + 1, q.push(v), p[v] = u;      // 3 lines in 1!
    }
    augment(t, INF);   // find the min edge weight 'f' in this path, if any
    if (f == 0) break; // we cannot send any more flow ('f' = 0), terminate
    mf += f;            // we can still send a flow, increase the max flow!
  }
  printf("%d\n", mf);                            // this is the max flow value
```

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

Source code: `ch4_08_edmonds_karp.cpp/java`

Exercise 4.6.3.1: Before continuing, answer the following question in Figure 4.26!

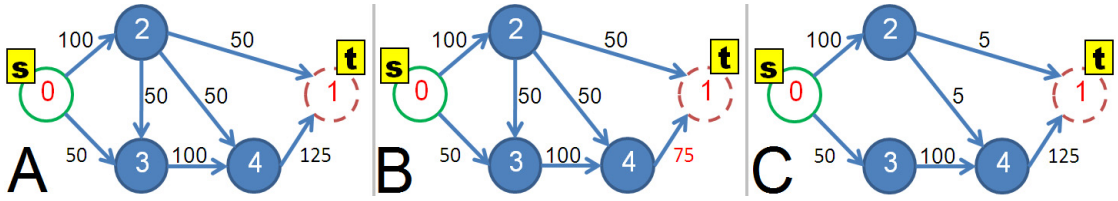


Figure 4.26: What are the Max Flow value of these three residual graphs?

Exercise 4.6.3.2: The main weakness of the simple code shown in this section is that enumerating the neighbors of a vertex takes $O(V)$ instead of $O(k)$ (where k is the number of neighbors of that vertex). The other (but not significant) weakness is that we also do not need `vi dist` as `bitset` (to flag whether a vertex has been visited or not) is sufficient. Modify the Edmonds Karp's code above so that it achieves its $O(VE^2)$ time complexity!

Exercise 4.6.3.3*: An even better implementation of the Edmonds Karp's algorithm is to avoid using the $O(V^2)$ Adjacency Matrix to store the residual capacity of each edge. A better way is to store both the original capacity and the actual flow (not just the residual) of each edge as an $O(V + E)$ modified Adjacency + Edge List. This way, we have three information for each edge: The original capacity of the edge, the flow currently in the edge, and we can derive the residual of an edge from the original capacity minus the flow of that edge. Now, modify the implementation again! How to handle the backward flow efficiently?

4.6.4 Flow Graph Modeling - Part 1

With the given Edmonds Karp's code above, solving a (basic/standard) Network Flow problem, especially Max Flow, is now simpler. It is now a matter of:

1. Recognizing that the problem is indeed a Network Flow problem
   (this will get better after you solve more Network Flow problems).
2. Constructing the appropriate flow graph (i.e. if using our code shown earlier:
   Initiate the residual matrix `res` and set the appropriate values for 's' and 't').
3. Running the Edmonds Karp's code on this flow graph.

In this subsection, we show an example of *modeling* the flow (residual) graph of UVa 259 - Software Allocation[17]. The abridged version of this problem is as follows: You are given up to 26 applications/apps (labeled 'A' to 'Z'), up to 10 computers (numbered from 0 to 9), the number of persons who brought in each application that day (one digit positive integer, or `[1..9]`), the list of computers that a particular application can run, and the fact that each computer can only run one application that day. Your task is to determine whether an allocation (that is, a *matching*) of applications to computers can be done, and if so, generates a possible allocation. If no, simply print an exclamation mark '!'.

One (bipartite) flow graph formulation is shown in Figure 4.27. We index the vertices from `[0..37]` as there are 26 + 10 + 2 special vertices = 38 vertices. The source s is given index 0, the 26 possible apps are given indices from `[1..26]`, the 10 possible computers are given indices from `[27..36]`, and finally the sink t is given index 37.

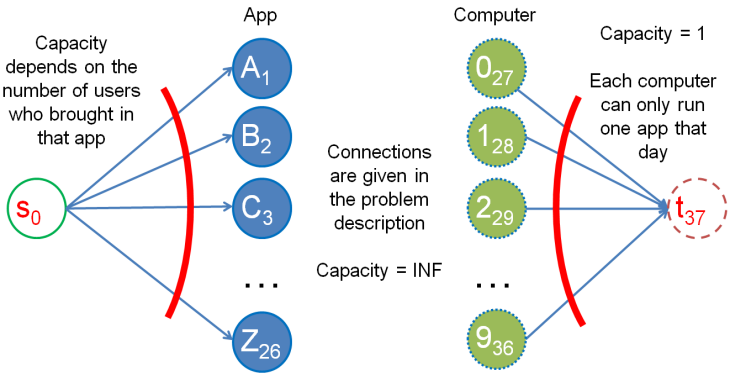


Figure 4.27: Residual Graph of UVa 259 [47]

[17] Actually this problem has small input size (we only have 26 + 10 = 36 vertices plus 2 more: source and sink) which make this problem still solvable with recursive backtracking (see Section 3.2). The name of this problem is 'assignment problem' or (special) bipartite matching with capacity.

Then, we link apps to computers as mentioned in the problem description. We link source s to all apps and link all computers to sink t. All edges in this flow graph are *directed* edges. The problem says that there can be *more than one* (say, X) users bringing in a particular app A in a given day. Thus, we set the edge weight (capacity) from source s to a particular app A to X. The problem also says that each computer can only be used once. Thus, we set the edge weight from each computer B to sink t to 1. The edge weight between apps to computers is set to ∞. With this arrangement, if there is a flow from an app A to a computer B and finally to sink t, that flow corresponds to *one matching* between that particular app A and computer B.

Once we have this flow graph, we can pass it to our Edmonds Karp's implementation shown earlier to obtain the Max Flow `mf`. If `mf` is equal to the number of applications brought in that day, then we have a solution, i.e. if we have X users bringing in app A, then X different paths (i.e. matchings) from A to sink t must be found by the Edmonds Karp's algorithm (and similarly for the other apps).

The actual app → computer assignments can be found by simply checking the backward edges from computers (vertices 27 - 36) to apps (vertices 1 - 26). A backward edge (computer → app) in the residual matrix `res` will contain a value +1 if the corresponding forward edge (app → computer) is selected in the paths that contribute to the Max Flow `mf`. This is also the reason why we start the flow graph with *directed* edges from apps to computers only.

Exercise 4.6.4.1: Why do we use ∞ for the edge weights (capacities) of directed edges from apps to computers? Can we use capacity 1 instead of ∞?

Exercise 4.6.4.2*: Is this kind of assignment problem (bipartite matching with capacity) can be solved with standard Max Cardinality Bipartite Matching (MCBM) algorithm shown later in Section 4.7.4? If it is possible, determine which one is the better solution?

4.6.5 Other Applications

There are several other interesting applications/variants of the problems involving flow in a network. We discuss three examples here while some others are deferred until Section 4.7.4 (Bipartite Graph), Section 9.13, Section 9.22, and Section 9.23. Note that some tricks shown here may also be applicable to other graph problems.

Minimum Cut

Let's define an s-t cut C = (S-component, T-component) as a partition of $V \in G$ such that source $s \in S$-component and sink $t \in T$-component. Let's also define a *cut-set* of C to be the set $\{(u, v) \in E \mid u \in S\text{-component}, v \in T\text{-component}\}$ such that if all edges in the cut-set of C are removed, the Max Flow from s to t is 0 (i.e. s and t are disconnected). The cost of an s-t cut C is defined by the sum of the capacities of the edges in the cut-set of C. The Minimum Cut problem, often abbreviated as just Min Cut, is to minimize the amount of capacity of an s-t cut. This problem is more general than finding bridges (see Section 4.2.1), i.e. in this case we can cut *more* than just one edge and we want to do so in the least cost way. As with bridges, Min Cut has applications in 'sabotaging' networks, e.g. One pure Min Cut problem is UVa 10480 - Sabotage.

The solution is simple: The by-product of computing Max Flow is Min Cut! Let's see Figure 4.24.D again. After Max Flow algorithm stops, we run graph traversal (DFS/BFS) from source s again. All reachable vertices from source s using positive weighted edges in the residual graph belong to the S-component (i.e. vertex 0 and 2). All other unreachable

vertices belong to the T-component (i.e. vertex 1 and 3). All edges connecting the S-component to the T-component belong to the cut-set of C (edge 0-3 (capacity 30/flow 30/residual 0), 2-3 (5/5/0) and 2-1 (25/25/0) in this case). The Min Cut value is 30+5+25 = 60 = Max Flow value `mf`. This is the minimum over all possible s-t cuts value.

Multi-source/Multi-sink

Sometimes, we can have more than one source and/or more than one sink. However, this variant is no harder than the original Network Flow problem with a single source and a single sink. Create a super source ss and a super sink st. Connect ss with all s with infinite capacity and also connect all t with st with infinite capacity, then run Edmonds Karp's as per normal. Note that we have seen this variant in **Exercise 4.4.2.1**.

Vertex Capacities

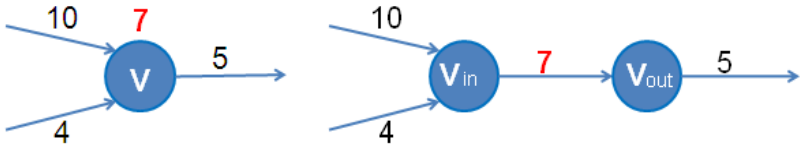


Figure 4.28: Vertex Splitting Technique

We can also have a Network Flow variant where the capacities are not just defined along the edges but *also on the vertices*. To solve this variant, we can use *vertex splitting* technique which (unfortunately) *doubles* the number of vertices in the flow graph. A weighted graph with a vertex weight can be converted into a more familiar one *without* vertex weight by splitting each weighted vertex v to v_{in} and v_{out}, reassigning its incoming/outgoing edges to v_{in}/v_{out}, respectively and finally putting the original vertex v's weight as the weight of edge $v_{in} \rightarrow v_{out}$. See Figure 4.28 for illustration. Now with all weights defined on edges, we can run Edmonds Karp's as per normal.

4.6.6 Flow Graph Modeling - Part 2

The hardest part of dealing with Network Flow problem is the modeling of the flow graph (assuming that we already have a good pre-written Max Flow code). In Section 4.6.4, we have seen one example modeling to deal with the assignment problem (or bipartite matching with capacity). Here, we present another (harder) flow graph modeling for UVa 11380 - Down Went The Titanic. Our advice before you continue reading: Please do not just memorize the solution but also try to understand the key steps to derive the required flow graph.

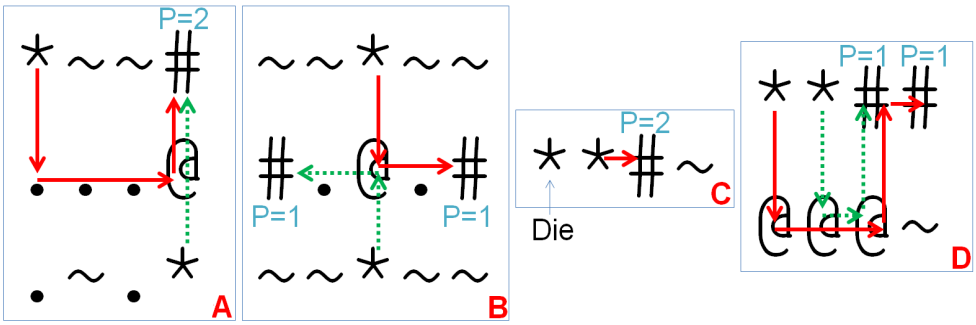


Figure 4.29: Some Test Cases of UVa 11380

In Figure 4.29, we have four small test cases of UVa 11380. You are given a small 2D grid containing these five characters as shown in Table 4.5. You want to put as many '*' (people) as possible to the (various) safe place(s): the '#' (large wood). The solid and dotted arrows in Figure 4.29 denotes the answer.

Symbol	Meaning	# Usage	Capacity
*	People staying on floating ice	1	1
~	Freezing water	0	0
.	Floating ice	1	1
@	Large iceberg	∞	1
#	Large wood	∞	P

Table 4.5: Characters Used in UVa 11380

To model the flow graph, we use the following thinking steps. In Figure 4.30.A, we first connect non '~' cells together with large capacity (1000 is enough for this problem). This describes the possible movements in the grid. In Figure 4.30.B, we set vertex capacities of '*' and '.' cells to 1 to indicate that they can only be used *once*. Then, we set vertex capacities of '@' and '#' to a large value (1000) to indicate that they can be used *several times*. In Figure 4.30.C, we create a source vertex s and sink vertex t. Source s is linked to all '*' cells in the grid with capacity 1 to indicate that there is one person to be saved. All '#' cells in the grid are connected to sink t with capacity P to indicate that the large wood can be used P times. Now, the required answer—the number of survivor(s)—equals to the max flow value between source s and sink t of this flow graph. As the flow graph uses vertex capacities, we need to use the *vertex splitting* technique discussed earlier.

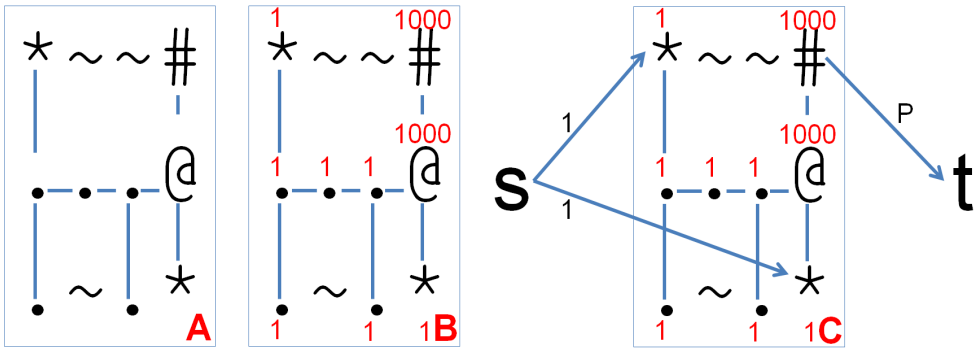


Figure 4.30: Flow Graph Modeling

Exercise 4.6.6.1*: Does $O(VE^2)$ Edmonds Karp's fast enough to compute the max flow value on the largest possible flow graph of UVa 11380: 30×30 grid and $P = 10$? Why?

Remarks About Network Flow in Programming Contests

As of 2013, when a Network (usually Max) Flow problem appears in a programming contest, it is *usually* one of the 'decider' problems. In ICPC, many interesting graph problems are written in such a way that they do not look like a Network Flow in a glance. The hardest part for the contestant is to realize that the underlying problem is indeed a Network Flow problem and able to model the flow graph correctly. This is the key skill that has to be mastered via practice.

To avoid wasting precious contest time coding the relatively long Max Flow library code, we suggest that in an ICPC team, one team member devotes significant effort in preparing a good Max Flow code (perhaps Dinic's algorithm implementation, see Section 9.7) and attempts various Network Flow problems available in many online judges to increase his/her familiarity towards Network Flow problems and its variants. In the list of programming exercises in this section, we have some simple Max Flow, bipartite matching with capacity (the assignment problem), Min Cut, and network flow problems involving vertex capacities. Try to solve as many programming exercises as possible.

In Section 4.7.4, we will see the classic Max Cardinality Bipartite Matching (MCBM) problem and see that this problem is also solvable with Max Flow. Later in Chapter 9, we will see some harder problems related to Network Flow, e.g. a faster Max Flow algorithm (Section 9.7), the Independent and Edge-Disjoint Paths problems (Section 9.13), the Max *Weighted* Independent Set on Bipartite Graph problem (Section 9.22), and the Min Cost (Max) Flow problem (Section 9.23).

In IOI, Network Flow (and its variants) is currently outside the 2009 syllabus [20]. So, IOI contestants can choose to skip this section. However, we believe that it is a good idea for IOI contestants to learn these more advanced material 'ahead of time' to improve your skills with graph problems.

Programming Exercises related to Network Flow:

- Standard Max Flow Problem (Edmonds Karp's)
  1. **UVa 00259 - Software Allocation *** (discussed in this section)
  2. **UVa 00820 - Internet Bandwidth *** (LA 5220, World Finals Orlando00, basic max flow, discussed in this section)
  3. UVa 10092 - The Problem with the ... (assignment problem, matching with capacity, similar with UVa 259)
  4. UVa 10511 - Councilling (matching, max flow, print the assignment)
  5. *UVa 10779 - Collectors Problem* (max flow modeling is not straightforward; the main idea is to build a flow graph such that each augmenting path corresponds to a series of exchange of duplicate stickers, starting with Bob giving away one of his duplicates, and ending with him receiving a new sticker; repeat until this is no longer possible)
  6. UVa 11045 - My T-Shirt Suits Me (assignment problem; but actually the input constraint is actually small enough for recursive backtracking)
  7. ***UVa 11167 - Monkeys in the Emei ... **** (max flow modeling; there are lots of edges in the flow graph; therefore, it is better to compress the capacity-1 edges whenever possible; use $O(V^2E)$ Dinic's max flow algorithm so that the high number of edges does not penalize the performance of your solution)
  8. UVa 11418 - Clever Naming Patterns (two layers of matching, it may be easier to use max flow solution)
- Variants
  1. UVa 10330 - Power Transmission (max flow with vertex capacities)
  2. UVa 10480 - Sabotage (straightforward min cut problem)
  3. ***UVa 11380 - Down Went The Titanic **** (discussed in this section)
  4. **UVa 11506 - Angry Programmer *** (min cut with vertex capacities)
  5. ***UVa 12125 - March of the Penguins **** (max flow modeling with vertex capacities; another interesting problem, similar level with UVa 11380)

4.7 Special Graphs

Some basic graph problems have simpler/faster polynomial algorithms if the given graph is *special*. Based on our experience, we have identified the following special graphs that commonly appear in programming contests: **Directed Acyclic Graph (DAG)**, **Tree**, **Eulerian Graph**, and **Bipartite Graph**. Problem authors may force the contestants to use specialized algorithms for these special graphs by giving a large input size to judge a correct algorithm for general graph as Time Limit Exceeded (TLE) (see a survey by [21]). In this section, we discuss some popular graph problems on these special graphs (see Figure 4.31)—many of which have been discussed earlier on general graphs. Note that at the time of writing, bipartite graph (Section 4.7.4) is still excluded in the IOI syllabus [20].

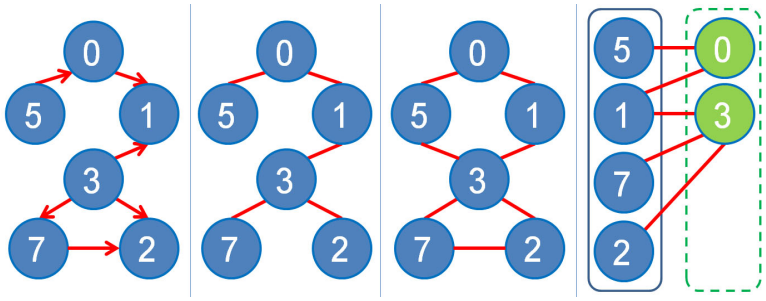


Figure 4.31: Special Graphs (L-to-R): DAG, Tree, Eulerian, Bipartite Graph

4.7.1 Directed Acyclic Graph

A Directed Acyclic Graph (DAG) is a special graph with the following characteristics: Directed and has no cycle. DAG guarantees the absence of cycle *by definition*. This makes problems that can be modeled as a DAG very suitable to be solved with Dynamic Programming (DP) techniques (see Section 3.5). After all, a DP recurrence must be *acyclic*. We can view DP states as vertices in an implicit DAG and the acyclic transitions between DP states as directed edges of that implicit DAG. Topological sort of this DAG (see Section 4.2.1) allows each overlapping subproblem (subgraph of the DAG) to be processed just once.

(Single-Source) Shortest/Longest Paths on DAG

The Single-Source Shortest Paths (SSSP) problem becomes much simpler if the given graph is a DAG. This is because a DAG has at least one topological order! We can use an $O(V+E)$ topological sort algorithm in Section 4.2.1 to find one such topological order, then relax the outgoing edges of these vertices according to this order. The topological order will ensure that if we have a vertex b that has an incoming edge from a vertex a, then vertex b is relaxed *after* vertex a has obtained correct shortest distance value. This way, the shortest distance values propagation is correct with just one $O(V+E)$ linear pass! This is also the essence of Dynamic Programming principle to avoid recomputation of overlapping subproblem covered earlier in Section 3.5. When we compute bottom-up DP, we essentially fill the DP table using the topological order of the underlying implicit DAG of DP recurrences.

The (Single-Source)[18] *Longest Paths* problem is a problem of finding the longest (simple[19]) paths from a starting vertex s to other vertices. The decision version of this problem

[18] Actually this can be multi-sources, as we can start from any vertex with 0 incoming degree.

[19] On general graph with positive weighted edges, the longest path problem is ill-defined as one can take a positive cycle and use that cycle to create an infinitely long path. This is the same issue as the negative cycle in shortest path problem. That is why for general graph, we use the term: 'longest *simple* path problem'. All paths in DAG are simple by definition so we can just use the term 'longest path problem'.

is NP-complete on a general graph[20]. However the problem is again easy if the graph has no cycle, which is true in a DAG. The solution for the Longest Paths on DAG[21] is just a minor tweak from the DP solution for the SSSP on DAG shown above. One trick is to multiply all edge weights by -1 and run the same SSSP solution as above. Finally, negate the resulting values to get the actual results.

The Longest Paths on DAG has applications in project scheduling, e.g. UVa 452 - Project Scheduling about Project Evaluation and Review Technique (PERT). We can model sub projects dependency as a DAG and the time needed to complete a sub project as *vertex weight*. The shortest possible time to finish the entire project is determined by the longest path in this DAG (a.k.a. the *critical* path) that starts from any vertex (sub project) with 0 incoming degree. See Figure 4.32 for an example with 6 sub projects, their estimated completion time units, and their dependencies. The longest path $0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 5$ with 16 time units determines the shortest possible time to finish the whole project. In order to achieve this, all sub projects along the longest (critical) path must be on time.

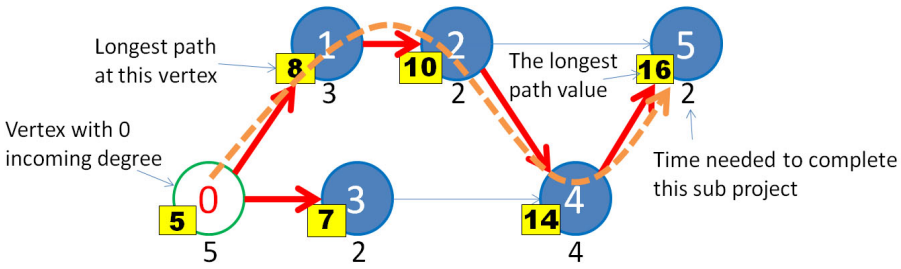


Figure 4.32: The Longest Path on this DAG

Counting Paths in DAG

Motivating problem (UVa 988 - Many paths, one destination): In life, one has many paths to choose, leading to many different lives. Enumerate how many different lives one can live, given a specific set of choices at each point in time. One is given a list of events, and a number of choices that can be selected, for each event. The objective is to count how many ways to go from the event that started it all (birth, index 0) to an event where one has no further choices (that is, death, index n).

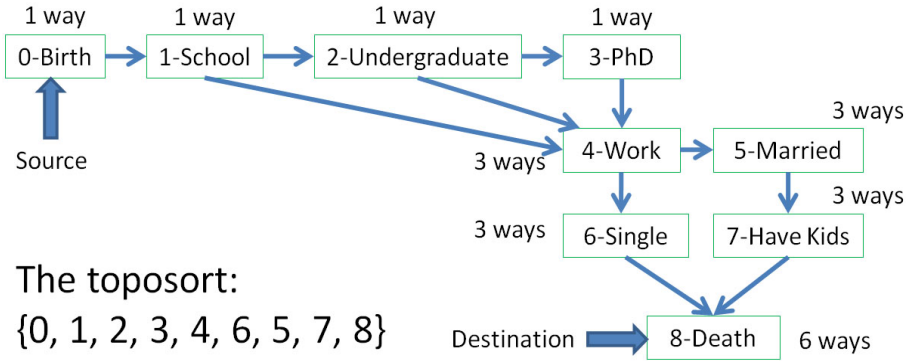


Figure 4.33: Example of Counting Paths in DAG - Bottom-Up

Clearly the underlying graph of the problem above is a DAG as one can move forward in time, but cannot go backward. The number of such paths can be found easily by computing one (any) topological order in $O(V + E)$ (in this problem, vertex 0/birth will always be the

[20]The decision version of this problem asks if the general graph has a simple path of total weight $\geq k$.
[21]The LIS problem in Section 3.5.2 can also be modeled as finding the Longest Paths on implicit DAG.

first in the topological order and the vertex n/death will always be the last in the topological order). We start by setting `num_paths[0] = 1`. Then, we process the remaining vertices one by one according to the topological order. When processing a vertex `u`, we update each neighbor `v` of `u` by setting `num_paths[v] += num_paths[u]`. After such $O(V + E)$ steps, we will know the number of paths in `num_paths[n]`. Figure 4.33 shows an example with 9 events and eventually 6 different possible life scenarios.

Bottom-Up versus Top-Down Implementations

Before we continue, we want to remark that all three solutions for shortest/longest/counting paths on/in DAG above are Bottom-Up DP solutions. We start from known base case(s) (the source vertex/vertices) and then we use topological order of the DAG to propagate the correct information to neighboring vertices without ever needing to backtrack.

We have seen in Section 3.5 that DP can also be written in Top-Down fashion. Using UVa 988 as an illustration, we can also write the DP solution as follows: Let `numPaths(i)` be the number of paths starting from vertex i to destination n. We can write the solution using this Complete Search recurrence relations:

1. `numPaths(n) = 1 // at destination n, there is only one possible path`
2. `numPaths(i) =` $\sum_j$ `numPaths(j), ∀j adjacent to i`

To avoid recomputations, we memoize the number of paths for each vertex i. There are $O(V)$ distinct vertices (states) and each vertex is only processed once. There are $O(E)$ edges and each edge is also visited at most once. Therefore the time complexity of this Top-Down approach is also $O(V + E)$, same as the Bottom-Up approach shown earlier. Figure 4.34 shows the similar DAG but the values are computed from destination to source (follow the dotted back arrows). Compare this Figure 4.34 with the previous Figure 4.33 where the values are computed from source to destination.

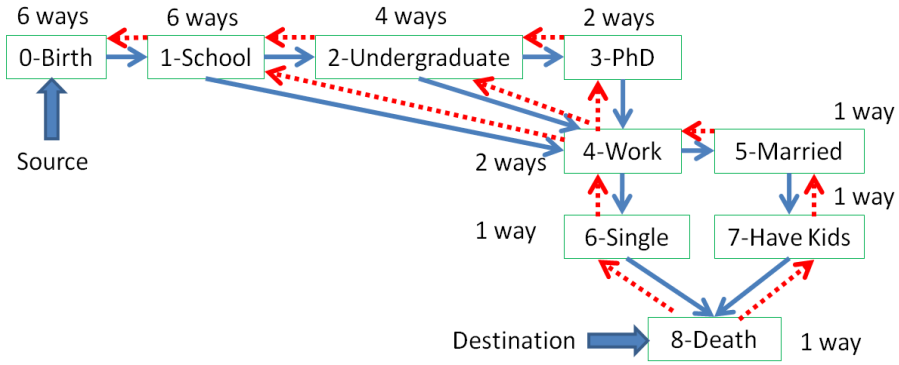


Figure 4.34: Example of Counting Paths in DAG - Top-Down

Converting General Graph to DAG

Sometimes, the given graph in the problem statement is not an *explicit DAG*. However, after further understanding, the given graph can be modeled as a DAG if we add one (or more) parameter(s). Once you have the DAG, the next step is to apply Dynamic Programming technique (either Top-Down or Bottom-Up). We illustrate this concept with two examples.

1. SPOJ 0101: Fishmonger

Abridged problem statement: Given the number of cities $3 \leq n \leq 50$, available time $1 \leq t \leq 1000$, and two $n \times n$ matrices (one gives travel times and another gives tolls between two cities), choose a route from the port city (vertex 0) in such a way that the fishmonger has to

pay as little tolls as possible to arrive at the market city (vertex $n-1$) within a certain time t. The fishmonger does *not* have to visit all cities. Output two information: The total tolls that is actually used and the actual traveling time. See Figure 4.35—left, for the original input graph of this problem.

Notice that there are *two* potentially conflicting requirements in this problem. The first requirement is to *minimize* tolls along the route. The second requirement is to *ensure* that the fishmonger arrive in the market city within allocated time, which may cause him to pay higher tolls in some part along the path. The second requirement is a *hard* constraint for this problem. That is, we must satisfy it, otherwise we do not have a solution.

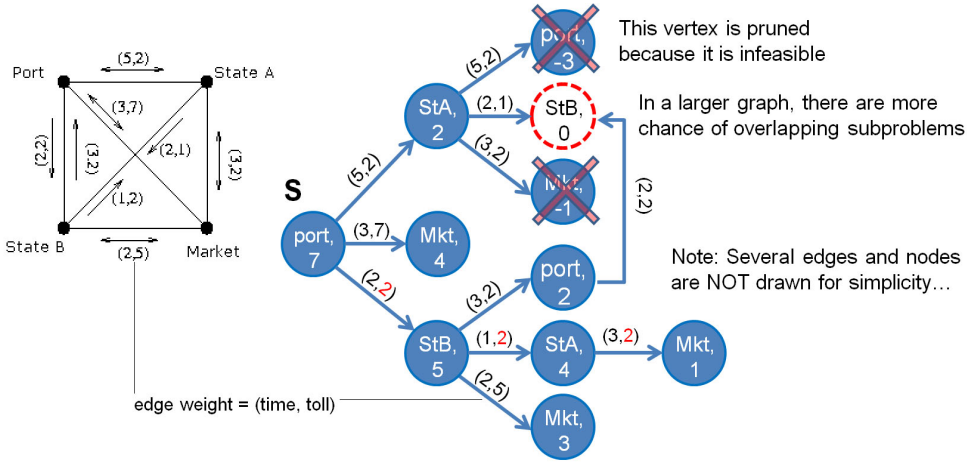


Figure 4.35: The Given General Graph (left) is Converted to DAG

Greedy SSSP algorithm like Dijkstra's (see Section 4.4.3)—on its pure form—does not work for this problem. Picking a path with the shortest travel time to help the fishmonger to arrive at market city $n-1$ using time $\leq t$ may not lead to the smallest possible tolls. Picking path with the cheapest tolls may not ensure that the fishmonger arrives at market city $n-1$ using time $\leq t$. These two requirements are not independent!

However, if we attach a parameter: `t_left` (time left) to each vertex, then the given graph turns into a DAG as shown in Figure 4.35, right. We start with a vertex `(port, t)` in the DAG. Every time the fishmonger moves from a current city `cur` to another city `X`, we move to a modified vertex `(X, t - travelTime[cur][X])` in the DAG via edge with weight `toll[cur][X]`. As time is a diminishing resource, we will never encounter a cyclic situation. We can then use this (Top-Down) DP recurrence: `go(cur, t_left)` to find the shortest path (in terms of total tolls paid) on this DAG. The answer can be found by calling `go(0, t)`. The C++ code of `go(cur, t_left)` is shown below:

```
ii go(int cur, int t_left) {       // returns a pair (tollpaid, timeneeded)
  if (t_left < 0) return ii(INF, INF);              // invalid state, prune
  if (cur == n - 1) return ii(0, 0); // at market, tollpaid=0, timeneeded=0
  if (memo[cur][t_left] != ii(-1, -1)) return memo[cur][t_left];
  ii ans = ii(INF, INF);
  for (int X = 0; X < n; X++) if (cur != X) {         // go to another city
    ii nextCity = go(X, t_left - travelTime[cur][X]);     // recursive step
    if (nextCity.first + toll[cur][X] < ans.first) {   // pick the min cost
      ans.first  = nextCity.first  + toll[cur][X];
      ans.second = nextCity.second + travelTime[cur][X];
  } }
  return memo[cur][t_left] = ans; }       // store the answer to memo table
```

Notice that by using Top-Down DP, we do not have to explicitly build the DAG and compute the required topological order. The recursion will do these steps for us. There are only $O(nt)$ distinct states (notice that the memo table is a pair object). Each state can be computed in $O(n)$. The overall time complexity is thus $O(n^2t)$—do-able.

2. Minimum Vertex Cover (on a Tree)

The tree data structure is also an acyclic data structure. But unlike DAG, there are no overlapping subtrees in a tree. Thus, there is no point of using Dynamic Programming (DP) technique on a standard tree. However, similar with the Fishmonger example above, some trees in programming contest problems turn into DAGs if we attach one (or more) parameter(s) to each vertex of the tree. Then, the solution is usually to run DP on the resulting DAG. Such problems are (inappropriately[22]) named as the 'DP on Tree' problems in competitive programming terminology.

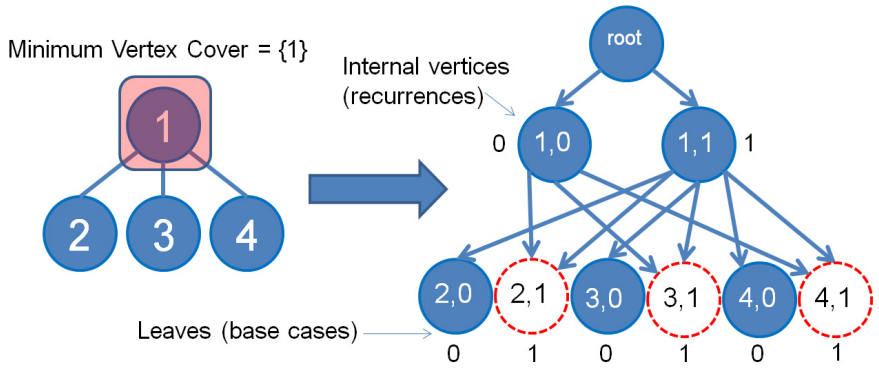


Figure 4.36: The Given General Graph/Tree (left) is Converted to DAG

An example of this DP on Tree problem is the problem of finding the Minimum Vertex Cover (MVC) on a Tree. In this problem, we have to select the smallest possible set of vertices $C \in V$ such that each edge of the tree is incident to at least one vertex of the set C. For the sample tree shown in Figure 4.36—left, the solution is to take vertex 1 only, because all edges 1-2, 1-3, 1-4 are all incident to vertex 1.

Now, there are only two possibilities for each vertex. Either it is taken, or it is not. By attaching this 'taken or not taken' status to each vertex, we convert the tree into a DAG (see Figure 4.36—right). Each vertex now has (vertex number, boolean flag taken/not). The implicit edges are determined with the following rules: 1). If the current vertex is not taken, then we have to take all its children to have a valid solution. 2). If the current vertex is taken, then we take the best between taking or not taking its children. We can now write this top down DP recurrences: `MVC(v, flag)`. The answer can be found by calling `min(MVC(root, false), MVC(root, true))`. Notice the presence of overlapping subproblems (dotted circles) in the DAG. However, as there are only $2 \times V$ states and each vertex has at most two incoming edges, this DP solution runs in $O(V)$.

```
int MVC(int v, int flag) {                              // Minimum Vertex Cover
  int ans = 0;
  if (memo[v][flag] != -1) return memo[v][flag];              // top down DP
  else if (leaf[v])      // leaf[v] is true if v is a leaf, false otherwise
    ans = flag;                                             // 1/0 = taken/not
```

[22]We have mentioned that there is no point of using DP on a Tree. But the term 'DP on Tree' that actually refers to 'DP on implicit DAG' is already a well-known term in competitive programming community.

```
  else if (flag == 0) {     // if v is not taken, we must take its children
    ans = 0;     // Note: 'Children' is an Adjacency List that contains the
    // directed version of the tree (parent points to its children; but the
    // children does not point to parents)
    for (int j = 0; j < (int)Children[v].size(); j++)
      ans += MVC(Children[v][j], 1);
  }
  else if (flag == 1) {          // if v is taken, take the minimum between
    ans = 1;                           // taking or not taking its children
    for (int j = 0; j < (int)Children[v].size(); j++)
      ans += min(MVC(Children.[v][j], 1), MVC(Children[v][j], 0));
  }
  return memo[v][flag] = ans;
}
```

Section 3.5—Revisited

Here, we want to re-highlight to the readers the strong linkage between DP techniques shown in Section 3.5 and algorithms on DAG. Notice that all programming exercises about shortest/longest/counting paths on/in DAG (or on general graph that is converted to DAG by some graph modeling/transformation) can also be classified under DP category. Often when we have a problem with DP solution that 'minimizes this', 'maximizes that', or 'counts something', that DP solution actually computes the shortest, the longest, or count the number of paths on/in the (usually implicit) DP recurrence DAG of that problem, respectively.

We now invite the readers to revisit some DP problems that we have seen earlier in Section 3.5 with this likely new viewpoint (viewing DP as algorithms on DAG is not commonly found in other Computer Science textbooks). As a start, we revisit the classic Coin Change problem. Figure 4.37 shows the same test case used in Section 3.5.2. There are $n = 2$ coin denominations: $\{1, 5\}$. The target amount is $V = 10$. We can model each vertex as the current value. Each vertex v has $n = 2$ unweighted edges that goes to vertex $v - 1$ and $v - 5$ in this test case, unless if it causes the index to go negative. Notice that the graph is a DAG and some states (highlighted with dotted circles) are overlapping (have more than one incoming edges). Now, we can solve this problem by finding the *shortest path* on this DAG from source $V = 10$ to target $V = 0$. The easiest topological order is to process the vertices in reverse sorted order, i.e. $\{10, 9, 8, \ldots, 1, 0\}$ is a valid topological order. We can definitely use the $O(V + E)$ shortest paths on DAG solution. However, since the graph is unweighted, we can also use the $O(V + E)$ BFS to solve this problem (using Dijkstra's is also possible but overkill). The path: $10 \rightarrow 5 \rightarrow 0$ is the shortest with total weight = 2 (or 2 coins needed). Note: For this test case, a greedy solution for coin change will also pick the same path: $10 \rightarrow 5 \rightarrow 0$.

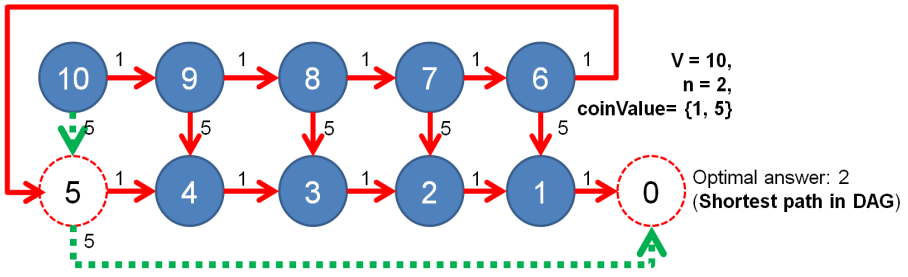


Figure 4.37: Coin Change as Shortest Paths on DAG

Next, let's revisit the classic 0-1 Knapsack Problem. This time we use the following test case: $n = 5, V = \{4, 2, 10, 1, 2\}, W = \{12, 1, 4, 1, 2\}, S = 15$. We can model each vertex as a pair of values `(id, remW)`. Each vertex has at least one edge `(id, remW)` to `(id+1, remW)` that corresponds to not taking a certain item `id`. Some vertices have edge `(id, remW)` to `(id+1, remW-W[id])` if `W[id]` $\leq$ `remW` that corresponds to taking a certain item `id`. Figure 4.38 shows some parts of the computation DAG of the standard 0-1 Knapsack Problem using the test case above. Notice that some states can be visited with more than one path (an overlapping subproblem is highlighted with a dotted circle). Now, we can solve this problem by finding the *longest path* on this DAG from the source `(0, 15)` to target `(5, any)`. The answer is the following path: `(0, 15)` $\rightarrow$ `(1, 15)` $\rightarrow$ `(2, 14)` $\rightarrow$ `(3, 10)` $\rightarrow$ `(4, 9)` $\rightarrow$ `(5, 7)` with weight $0 + 2 + 10 + 1 + 2 = 15$.

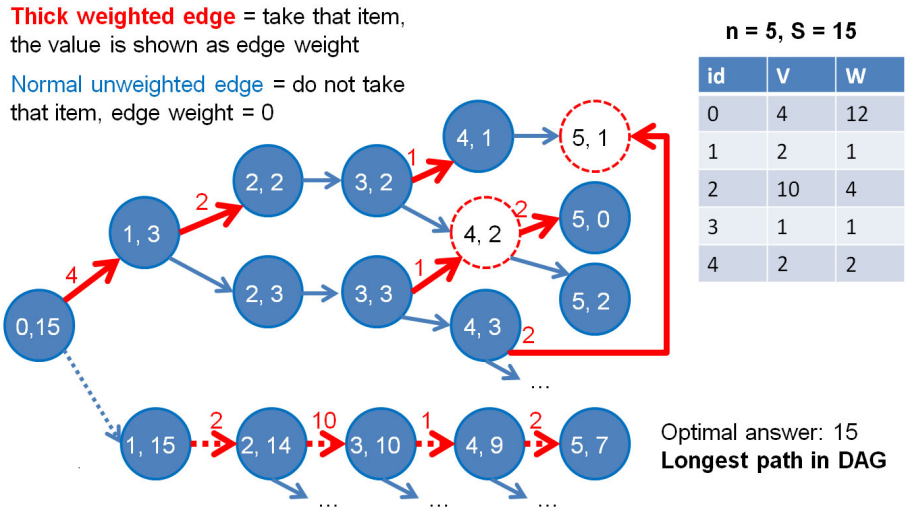


Figure 4.38: 0-1 Knapsack as Longest Paths on DAG

Let's see one more example: The solution for UVa 10943 - How do you add? discussed in Section 3.5.3. If we draw the DAG of this test case: $n = 3, K = 4$, then we have a DAG as shown in Figure 4.39. There are overlapping subproblems highlighted with dotted circles. If we count the number of paths in this DAG, we will indeed find the answer = 20 paths.

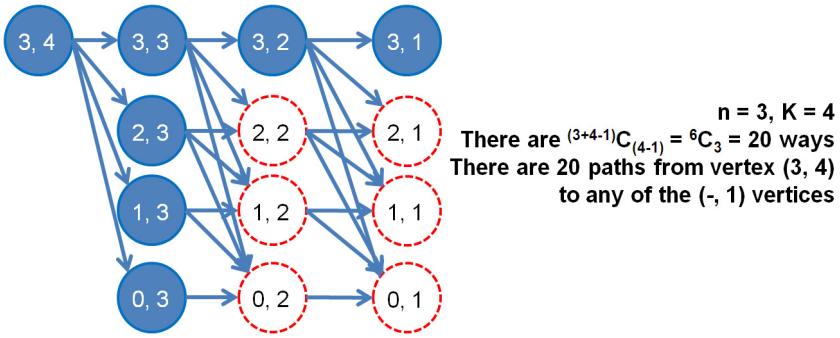


Figure 4.39: UVa 10943 as Counting Paths in DAG

Exercise 4.7.1.1*: Draw the DAG for some test cases of the other classical DP problems not mentioned above: Traveling Salesman Problem (TSP) $\approx$ shortest paths on the implicit DAG, Longest Increasing Subsequence (LIS) $\approx$ longest paths of the implicit DAG, Counting Change variant (the one about counting the number of possible ways to get value V cents using a list of denominations of N coins) $\approx$ counting paths in DAG, etc.

4.7.2 Tree

Tree is a special graph with the following characteristics: It has $E = V$-1 (any $O(V + E)$ algorithm on tree is $O(V)$), it has no cycle, it is connected, and there exists one unique path for any pair of vertices.

Tree Traversal

In Section 4.2.1 and 4.2.2, we have seen $O(V + E)$ DFS and BFS algorithms for traversing a general graph. If the given graph is a *rooted binary tree*, there are *simpler* tree traversal algorithms like pre-order, in-order, and post-order traversal (note: level-order traversal is essentially BFS). There is no major time speedup as these tree traversal algorithms also run in $O(V)$, but the code are simpler. Their pseudo-code are shown below:

```
pre-order(v)                in-order(v)                post-order(v)
  visit(v);                   in-order(left(v));         post-order(left(v));
  pre-order(left(v));         visit(v);                  post-order(right(v));
  pre-order(right(v));        in-order(right(v));        visit(v);
```

Finding Articulation Points and Bridges in Tree

In Section 4.2.1, we have seen $O(V + E)$ Tarjan's DFS algorithm for finding articulation points and bridges of a graph. However, if the given graph is a tree, the problem becomes simpler: All edges on a tree are bridges and all internal vertices (degree > 1) are articulation points. This is still $O(V)$ as we have to scan the tree to count the number of internal vertices, but the code is *simpler*.

Single-Source Shortest Paths on Weighted Tree

In Sections 4.4.3 and 4.4.4, we have seen two general purpose algorithms ($O((V + E) \log V)$ Dijkstra's and $O(VE)$ Bellman-Ford's) for solving the SSSP problem on a weighted graph. But if the given graph is a weighted tree, the SSSP problem becomes *simpler*: Any $O(V)$ graph traversal algorithm, i.e. BFS or DFS, can be used to solve this problem. There is only one unique path between any two vertices in a tree, so we simply traverse the tree to find the unique path connecting the two vertices. The shortest path weight between these two vertices is basically the sum of edge weights of this unique path (e.g. from vertex `5` to vertex `3` in Figure 4.40.A, the unique path is `5->0->1->3` with weight `4+2+9 = 15`).

All-Pairs Shortest Paths on Weighted Tree

In Section 4.5, we have seen a general purpose algorithm ($O(V^3)$ Floyd Warshall's) for solving the APSP problem on a weighted graph. However, if the given graph is a weighted tree, the APSP problem becomes *simpler*: Repeat the SSSP on weighted tree V times, setting each vertex as the source vertex one by one. The overall time complexity is $O(V^2)$.

Diameter of Weighted Tree

For general graph, we need $O(V^3)$ Floyd Warshall's algorithm discussed in Section 4.5 plus another $O(V^2)$ all-pairs check to compute the diameter. However, if the given graph is a weighted tree, the problem becomes *simpler*. We only need two $O(V)$ traversals: Do DFS/BFS from *any* vertex s to find the furthest vertex x (e.g. from vertex `s=1` to vertex `x=2` in Figure 4.40.B1), then do DFS/BFS one more time from vertex x to get the true

furthest vertex y from x. The length of the unique path along x to y is the diameter of that tree (e.g. path `x=2->3->1->0->y=5` with length 20 in Figure 4.40.B2).

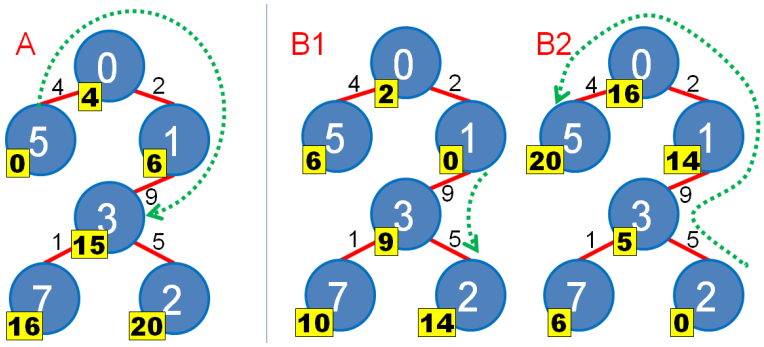


Figure 4.40: A: SSSP (Part of APSP); B1-B2: Diameter of Tree

Exercise 4.7.2.1*: Given the inorder and preorder traversal of a rooted Binary Search Tree (BST) T containing n vertices, write a recursive pseudo-code to output the postorder traversal of that BST. What is the time complexity of your best algorithm?

Exercise 4.7.2.2*: There is an even faster solution than $O(V^2)$ for the All-Pairs Shortest Paths problem on Weighted Tree. It uses LCA. How?

4.7.3 Eulerian Graph

An *Euler path* is defined as a path in a graph which visits *each edge* of the graph *exactly once*. Similarly, an *Euler tour/cycle* is an Euler path which starts and ends on the same vertex. A graph which has either an Euler path or an Euler tour is called an Eulerian graph[23].

This type of graph is first studied by Leonhard *Euler* while solving the Seven Bridges of Königsberg problem in 1736. Euler's finding 'started' the field of graph theory!

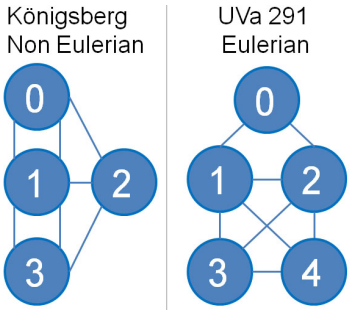


Figure 4.41: Eulerian

Eulerian Graph Check

To check whether a connected undirected graph has an Euler tour is simple. We just need to check if all its vertices have even degrees. It is similar for the Euler path, i.e. an undirected graph has an Euler path if all except two vertices have even degrees. This Euler path will start from one of these odd degree vertices and end in the other[24]. Such degree check can be done in $O(V + E)$, usually done simultaneously when reading the input graph. You can try this check on the two graphs in Figure 4.41.

Printing Euler Tour

While checking whether a graph is Eulerian is easy, finding the actual Euler tour/path requires more work. The code below produces the desired Euler tour when given an unweighted Eulerian graph stored in an Adjacency List where the second attribute in edge information pair is a Boolean 1 (this edge can still be used) or 0 (this edge can no longer be used).

[23]Compare this property with the *Hamiltonian path/cycle* in TSP (see Section 3.5.2).

[24]Euler path on *directed graph* is also possible: Graph must be connected, has equal in/outdegree vertices, at most one vertex with indegree - outdegree = 1, and at most one vertex with outdegree - indegree = 1.

```
list<int> cyc;             // we need list for fast insertion in the middle

void EulerTour(list<int>::iterator i, int u) {
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (v.second) {           // if this edge can still be used/not removed
      v.second = 0;     // make the weight of this edge to be 0 ('removed')
      for (int k = 0; k < (int)AdjList[v.first].size(); k++) {
        ii uu = AdjList[v.first][k];          // remove bi-directional edge
        if (uu.first == u && uu.second) {
          uu.second = 0;
          break;
      } }
      EulerTour(cyc.insert(i, u), v.first);
} } }

// inside int main()
  cyc.clear();
  EulerTour(cyc.begin(), A);    // cyc contains an Euler tour starting at A
  for (list<int>::iterator it = cyc.begin(); it != cyc.end(); it++)
    printf("%d\n", *it);                                  // the Euler tour
```

4.7.4 Bipartite Graph

Bipartite Graph is a special graph with the following characteristics: The set of vertices V can be partitioned into two disjoint sets V_1 and V_2 and all edges in $(u, v) \in E$ has the property that $u \in V_1$ and $v \in V_2$. This makes a Bipartite Graph free from odd-length cycles (see **Exercise 4.2.6.3**). Note that Tree is also a Bipartite Graph!

Max Cardinality Bipartite Matching (MCBM) and Its Max Flow Solution

Motivating problem (from TopCoder Open 2009 Qualifying 1 [31]): Given a list of numbers N, return a list of all the elements in N that can be paired with $N[0]$ successfully as part of a *complete prime pairing*, sorted in ascending order. Complete prime pairing means that each element a in N is paired to a unique other element b in N such that $a + b$ is prime.

For example: Given a list of numbers $N = \{1, 4, 7, 10, 11, 12\}$, the answer is $\{4, 10\}$. This is because pairing $N[0] = 1$ with 4 results in a prime pair and the other four items can also form two prime pairs ($7 + 10 = 17$ and $11 + 12 = 23$). Similar situation by pairing $N[0] = 1$ with 10, i.e. $1 + 10 = 11$ is a prime pair and we also have two other prime pairs ($4 + 7 = 11$ and $11 + 12 = 23$). We cannot pair $N[0] = 1$ with any other item in N. For example, if we pair $N[0] = 1$ with 12, we have a prime pair but there will be no way to pair the remaining 4 numbers to form 2 more prime pairs.

Constraints: List N contains an even number of elements (`[2..50]`). Each element of N will be between `[1..1000]`. Each element of N will be distinct.

Although this problem involves prime numbers, it is not a pure math problem as the elements of N are not more than 1K—there are not too many primes below 1000 (only 168 primes). The issue is that we cannot do Complete Search pairings as there are ${}_{50}C_2$ possibilities for the first pair, ${}_{48}C_2$ for the second pair, ..., until ${}_2C_2$ for the last pair. DP with bitmask technique (Section 8.3.1) is also not usable because 2^{50} is too big.

The key to solve this problem is to realize that this pairing (matching) is done on *bipartite graph*! To get a prime number, we need to sum 1 odd + 1 even, because 1 odd + 1 odd (or 1 even + 1 even) produces an even number (which is not prime). Thus we can split odd/even numbers to `set1`/`set2` and add edge `i` → `j` if `set1[i] + set2[j]` is prime.

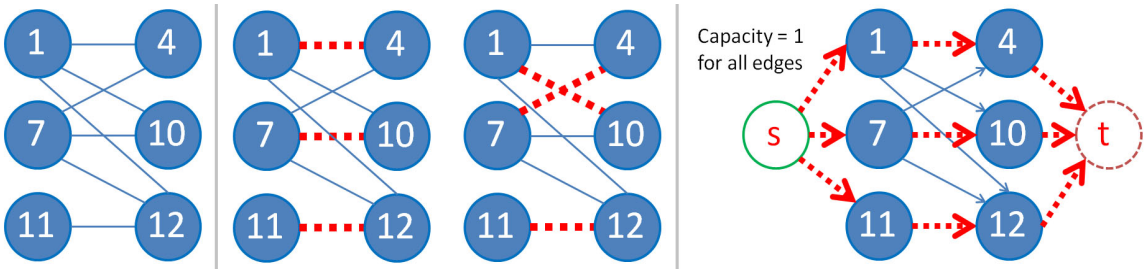


Figure 4.42: Bipartite Matching problem can be reduced to a Max Flow problem

After we build this bipartite graph, the solution is trivial: If the size of `set1` and `set2` are different, a complete pairing is not possible. Otherwise, if the size of both sets are $n/2$, try to match `set1[0]` with `set2[k]` for k = `[0..n/2-1]` and do Max Cardinality Bipartite Matching (MCBM) for the rest (MCBM is one of the most common applications involving Bipartite Graph). If we obtain $n/2 - 1$ more matchings, add `set2[k]` to the answer. For this test case, the answer is {4, 10} (see Figure 4.42, middle).

MCBM problem can be reduced to the Max Flow problem by assigning a dummy source vertex s connected to all vertices in `set1` and all vertices in `set2` are connected to a dummy sink vertex t. The edges are directed ($s \rightarrow u$, $u \rightarrow v$, $v \rightarrow t$ where $u \in$ `set1` and $v \in$ `set2`). By setting the capacities of all edges in this flow graph to 1, we force each vertex in `set1` to be matched with at most one vertex in `set2`. The Max Flow will be equal to the maximum number of matchings on the original graph (see Figure 4.42—right for an example).

Max Independent Set and Min Vertex Cover on Bipartite Graph

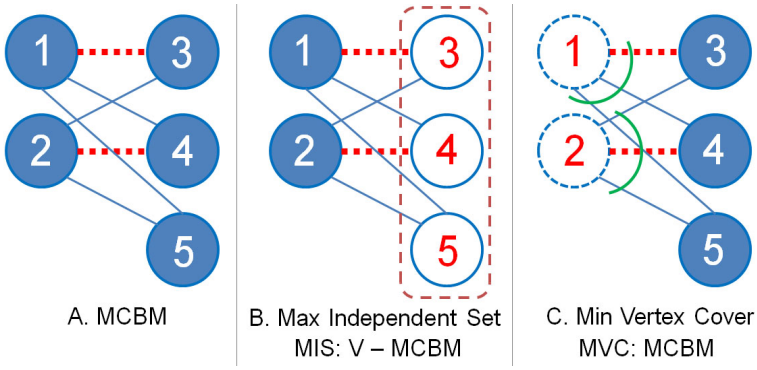


Figure 4.43: MCBM Variants

An Independent Set (IS) of a graph G is a subset of the vertices such that no two vertices in the subset represent an edge of G. A Max IS (MIS) is an IS such that adding any other vertex to the set causes the set to contain an edge. In Bipartite Graph, the size of the MIS + MCBM = V. Or in another word: MIS = V - MCBM. In Figure 4.43.B, we have a Bipartite Graph with 2 vertices on the left side and 3 vertices on the right side. The MCBM is 2 (two dashed lines) and the MIS is 5-2 = 3. Indeed, {3, 4, 5} are the members of the MIS of this Bipartite Graph. Another term for MIS is *Dominating Set*.

A vertex cover of a graph G is a set C of vertices such that each edge of G is incident to at least one vertex in C. In Bipartite Graph, the number of matchings in an MCBM equals the number of vertices in a Min Vertex Cover (MVC)—this is a theorem by a Hungarian mathematician Dénes *König*. In Figure 4.43.C, we have the same Bipartite Graph as earlier with MCBM = 2. The MVC is also 2. Indeed, {1, 2} are the members of the MVC of this Bipartite Graph.

We remark that although the MCBM/MIS/MVC values are unique, the solutions may not be unique. Example: In Figure 4.43.A, we can also match {1, 4} and {2, 5} with the same maximum cardinality of 2.

Sample Application: UVa 12083 - Guardian of Decency

Abridged problem description: Given $N \leq 500$ students (in terms of their height, gender, music style, and favorite sport), determine how many students are eligible for an excursion if the teacher wants any pair of two students satisfy at least one of these four criteria so that no pair of students becomes a couple: 1). Their height differs by more than 40 cm.; 2). They are of the same sex.; 3). Their preferred music style is different.; 4). Their favorite sport is the same (they are likely to be fans of different teams and that would result in fighting).

First, notice that the problem is about finding the Maximum Independent Set, i.e. the chosen students should not have any chance of becoming a couple. Independent Set is a hard problem in general graph, so let's check if the graph is special. Next, notice that there is an easy Bipartite Graph in the problem description: The gender of students (constraint number two). We can put the male students on the left side and the female students on the right side. At this point, we should ask: What should be the edges of this Bipartite Graph? The answer is related to the Independent Set problem: We draw an edge between a male student i and a female student j if there is a chance that (i, j) may become a couple.

In the context of this problem: If i and j have *DIFFERENT* gender *and* their height differs by *NOT MORE* than 40 cm *and* their preferred music style is *THE SAME and* their favorite sport is *DIFFERENT*, then this pair, one male student i and one female student j, has a high probability to be a couple. The teacher can only choose one of them.

Now, once we have this Bipartite Graph, we can run the MCBM algorithm and report: $N - MCBM$. With this example, we again re-highlighted the importance of having good *graph modeling* skill! There is no point knowing MCBM algorithm and its code if contestant cannot identify the Bipartite Graph from the problem description in the first place.

Augmenting Path Algorithm for Max Cardinality Bipartite Matching

There is a better way to solve the MCBM problem in programming contest (in terms of implementation time) rather than going via the 'Max Flow route'. We can use the specialized and easy to implement $O(VE)$ *augmenting path* algorithm. With its implementation handy, all the MCBM problems, including other graph problems that requires MCBM—like the Max Independent Set in Bipartite Graph, Min Vertex Cover in Bipartite Graph, and Min Path Cover on DAG (see Section 9.24)—can be easily solved.

An augmenting path is a path that starts from a *free (unmatched)* vertex on the left set of the Bipartite Graph, alternate between a free edge (now on the right set), a matched edge (now on the left set again), ..., a free edge (now on the right set) until the path finally arrives on a *free vertex* on the right set of the Bipartite Graph. A lemma by Claude *Berge* in 1957 states that a matching M in graph G is maximum (has the max possible number of edges) if and only if there is no more augmenting path in G. This augmenting path algorithm is a direct implementation of Berge's lemma: Find and then eliminate *augmenting paths*.

Now let's take a look at a simple Bipartite Graph in Figure 4.44 with n and m vertices on the left set and the right set, respectively. Vertices on the left set are numbered from `[1..n]` and vertices of the right set are numbered from `[n+1..n+m]`. This algorithm tries to find and then eliminates augmenting paths starting from free vertices on the left set.

We start with a free vertex 1. In Figure 4.44.A, we see that this algorithm will 'wrongly[25]' match vertex 1 with vertex 3 (rather than vertex 1 with vertex 4) as path 1-3 is already a simple augmenting path. Both vertex 1 and vertex 3 are free vertices. By matching vertex 1 and vertex 3, we have our first matching. Notice that after we match vertex 1 and 3, we are unable to find another matching.

In the next iteration (when we are in a free vertex 2), this algorithm now shows its full strength by finding the following augmenting path that starts from a free vertex 2 on the left, goes to vertex 3 via a free edge (2-3), goes to vertex 1 via a matched edge (3-1), and finally goes to vertex 4 via a free edge again (1-4). Both vertex 2 and vertex 4 are free vertices. Therefore, the augmenting path is 2-3-1-4 as seen in Figure 4.44.B and 4.44.C.

If we flip the edge status in this augmenting path, i.e. from 'free to matched' and 'matched to free', we will get *one more matching*. See Figure 4.44.C where we flip the status of edges along the augmenting path 2-3-1-4. The updated matching is reflected in Figure 4.44.D.

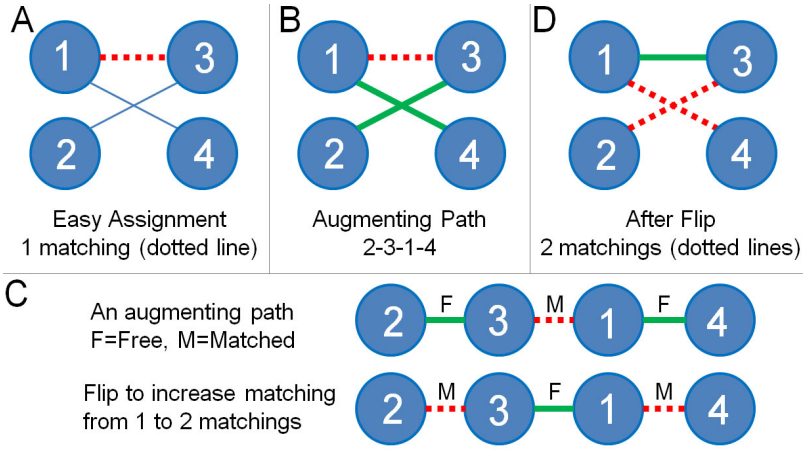


Figure 4.44: Augmenting Path Algorithm

This algorithm will keep doing this process of finding augmenting paths and eliminating them until there is no more augmenting path. As the algorithm repeats $O(E)$ DFS-like[26] code V times, it runs in $O(VE)$. The code is shown below. We remark that this is not the best algorithm for finding MCBM. Later in Section 9.12, we will learn Hopcroft Karp's algorithm that can solve the MCBM problem in $O(\sqrt{V}E)$ [28].

Exercise 4.7.4.1*: In Figure 4.42—right, we have seen a way to reduce an MCBM problem into a Max Flow problem. The question: Does the edges in the flow graph have to be directed? Is it OK if we use undirected edges in the flow graph?

Exercise 4.7.4.2*: List down common keywords that can be used to help contestants spot a bipartite graph in the problem statement! e.g. odd-even, male-female, etc.

Exercise 4.7.4.3*: Suggest a simple improvement for the augmenting path algorithm that can avoid its worst case $O(VE)$ time complexity on (near) complete bipartite graph!

---

[25]We assume that the neighbors of a vertex are ordered based on increasing vertex number, i.e. from vertex 1, we will visit vertex 3 first *before* vertex 4.

[26]To simplify the analysis, we assume that $E > V$ in such bipartite graphs.

```
vi match, vis;                                          // global variables

int Aug(int l) {                 // return 1 if an augmenting path is found
  if (vis[l]) return 0;                                 // return 0 otherwise
  vis[l] = 1;
  for (int j = 0; j < (int)AdjList[l].size(); j++) {
    int r = AdjList[l][j];  // edge weight not needed -> vector<vi> AdjList
    if (match[r] == -1 || Aug(match[r])) {
      match[r] = l; return 1;                           // found 1 matching
  } }
  return 0;                                                 // no matching
}
// inside int main()
  // build unweighted bipartite graph with directed edge left->right set
  int MCBM = 0;
  match.assign(V, -1);    // V is the number of vertices in bipartite graph
  for (int l = 0; l < n; l++) {                 // n = size of the left set
    vis.assign(n, 0);                         // reset before each recursion
    MCBM += Aug(l);
  }
  printf("Found %d matchings\n", MCBM);
```

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

Source code: ch4_09_mcbm.cpp/java

Remarks About Special Graphs in Programming Contests

Of the four special graphs mentioned in this Section 4.7. DAGs and Trees are more popular, especially for IOI contestants. It is *not* rare that Dynamic Programming (DP) on DAG or on tree appear as IOI task. As these DP variants (typically) have efficient solutions, the input size for them are usually large. The next most popular special graph is the Bipartite Graph. This special graph is suitable for Network Flow and Bipartite Matching problems. We reckon that contestants must master the usage of the simpler augmenting path algorithm for solving the Max Cardinality Bipartite Matching (MCBM) problem. We have seen in this section that many graph problems are somehow reduce-able to MCBM. ICPC contestants should be familiar with Bipartite Graph on top of DAG and Tree. IOI contestants do not have to worry with Bipartite Graph as it is still outside IOI 2009 syllabus [20]. The other special graph discussed in this chapter—the Eulerian Graph—does not have too many contest problems involving it nowadays. There are other possible special graphs, but we rarely encounter them, e.g. Planar Graph; Complete Graph K_n; Forest of Paths; Star Graph; etc. When they appear, try to utilize their special properties to speed up your algorithms.

Profile of Algorithm Inventors

Dénes König (1884-1944) was a Hungarian mathematician who worked in and wrote the first textbook on the field of graph theory. In 1931, König describes an equivalence between the Maximum Matching problem and the Minimum Vertex Cover problem in the context of Bipartite Graphs, i.e. he proves that MCBM = MVC in Bipartite Graph.

Claude Berge (1926-2002) was a French mathematician, recognized as one of the modern founders of combinatorics and graph theory. His main contribution that is included in this book is the Berge's lemma, which states that a matching M in a graph G is maximum if and only if there is no more augmenting path with respect to M in G.

Programming Exercises related to Special Graphs:

- Single-Source Shortest/Longest Paths on DAG
  1. UVa 00103 - Stacking Boxes (longest paths on DAG; backtracking OK)
  2. **UVa 00452 - Project Scheduling *** (PERT; longest paths on DAG; DP)
  3. UVa 10000 - Longest Paths (longest paths on DAG; backtracking OK)
  4. UVa 10051 - Tower of Cubes (longest paths on DAG; DP)
  5. UVa 10259 - Hippity Hopscotch (longest paths on implicit DAG; DP)
  6. **UVa 10285 - Longest Run ... *** (longest paths on implicit DAG; however, the graph is small enough for recursive backtracking solution)
  7. **UVa 10350 - Liftless Eme *** (shortest paths; implicit DAG; DP)
     Also see: Longest Increasing Subsequence (see Section 3.5.3)
- Counting Paths in DAG
  1. UVa 00825 - Walking on the Safe Side (counting paths in implicit DAG; DP)
  2. UVa 00926 - Walking Around Wisely (similar to UVa 825)
  3. UVa 00986 - How Many? (counting paths in DAG; DP; s: x, y, lastmove, peaksfound; t: try NE/SE)
  4. **UVa 00988 - Many paths, one ... *** (counting paths in DAG; DP)
  5. **UVa 10401 - Injured Queen Problem *** (counting paths in implicit DAG; DP; s: col, row; t: next col, avoid 2 or 3 adjacent rows)
  6. UVa 10926 - How Many Dependencies? (counting paths in DAG; DP)
  7. UVa 11067 - Little Red Riding Hood (similar to UVa 825)
  8. *UVa 11655 - Waterland* (counting paths in DAG and one more similar task: counting the number of vertices involved in the paths)
  9. **UVa 11957 - Checkers *** (counting paths in DAG; DP)
- Converting General Graph to DAG
  1. UVa 00590 - Always on the Run (s: pos, **day_left**)
  2. **UVa 00907 - Winterim Backpack... *** (s: pos, **night_left**)
  3. UVa 00910 - TV Game (s: pos, **move_left**)
  4. UVa 10201 - Adventures in Moving ... (s: pos, **fuel_left**)
  5. UVa 10543 - Traveling Politician (s: pos, **given_speech**)
  6. UVa 10681 - Teobaldo's Trip (s: pos, **day_left**)
  7. UVa 10702 - Traveling Salesman (s: pos, **T_left**)
  8. UVa 10874 - Segments (s: row, **left/right**; t: go left/right)
  9. **UVa 10913 - Walking ... *** (s: r, c, **neg_left**, **stat**; t: down/(left/right))
  10. UVa 11307 - Alternative Arborescence (Min Chromatic Sum, max 6 colors)
  11. **UVa 11487 - Gathering Food *** (s: row, col, **cur_food**, **len**; t: 4 dirs)
  12. UVa 11545 - Avoiding ... (s: cPos, **cTime**, **cWTime**; t: move forward/rest)
  13. UVa 11782 - Optimal Cut (s: id, **rem_K**; t: take root/try left-right subtree)
  14. SPOJ 0101 - Fishmonger (discussed in this section)

- Tree
  1. UVa 00112 - Tree Summing (backtracking)
  2. UVa 00115 - Climbing Trees (tree traversal, Lowest Common Ancestor)
  3. UVa 00122 - Trees on the level (tree traversal)
  4. UVa 00536 - Tree Recovery (reconstructing tree from pre + inorder)
  5. *UVa 00548 - Tree* (reconstructing tree from in + postorder traversal)
  6. UVa 00615 - Is It A Tree? (graph property check)
  7. UVa 00699 - The Falling Leaves (preorder traversal)
  8. UVa 00712 - S-Trees (simple binary tree traversal variant)
  9. UVa 00839 - Not so Mobile (can be viewed as recursive problem on tree)
  10. UVa 10308 - Roads in the North (diameter of tree, discussed in this section)
  11. ***UVa 10459 - The Tree Root **** (identify the diameter of this tree)
  12. UVa 10701 - Pre, in and post (reconstructing tree from pre + inorder)
  13. ***UVa 10805 - Cockroach Escape ... **** (involving diameter)
  14. *UVa 11131 - Close Relatives* (read tree; produce two postorder traversals)
  15. *UVa 11234 - Expressions* (converting post-order to level-order, binary tree)
  16. UVa 11615 - Family Tree (counting size of subtrees)
  17. **UVa 11695 - Flight Planning *** (cut the worst edge along the tree diameter, link two centers)
  18. *UVa 12186 - Another Crisis* (the input graph is a tree)
  19. *UVa 12347 - Binary Search Tree* (given pre-order traversal of a BST, use BST property to get the BST, output the post-order traversal that BST)
- Eulerian Graph
  1. UVa 00117 - The Postal Worker ... (Euler tour, cost of tour)
  2. UVa 00291 - The House of Santa ... (Euler tour, small graph, backtracking)
  3. **UVa 10054 - The Necklace *** (printing the Euler tour)
  4. UVa 10129 - Play on Words (Euler Graph property check)
  5. **UVa 10203 - Snow Clearing *** (the underlying graph is Euler graph)
  6. **UVa 10596 - Morning Walk *** (Euler Graph property check)
- Bipartite Graph:
  1. *UVa 00663 - Sorting Slides* (try disallowing an edge to see if MCBM changes; which implies that the edge has to be used)
  2. UVa 00670 - The Dog Task (MCBM)
  3. UVa 00753 - A Plug for Unix (initially a non standard matching problem but this problem can be reduced to a simple MCBM problem)
  4. UVa 01194 - Machine Schedule (LA 2523, Beijing02, Min Vertex Cover/MVC)
  5. UVa 10080 - Gopher II (MCBM)
  6. **UVa 10349 - Antenna Placement *** (Max Independent Set: V - MCBM)
  7. **UVa 11138 - Nuts and Bolts *** (pure MCBM problem, if you are new with MCBM, it is good to start from this problem)
  8. **UVa 11159 - Factors and Multiples *** (MIS, but ans is the MCBM)
  9. *UVa 11419 - SAM I AM* (MVC, König theorem)
  10. UVa 12083 - Guardian of Decency (LA 3415, NorthwesternEurope05, MIS)
  11. UVa 12168 - Cat vs. Dog (LA 4288, NorthwesternEurope08, MIS)
  12. Top Coder Open 2009: Prime Pairs (discussed in this section)

4.8 Solution to Non-Starred Exercises

Exercise 4.2.2.1: Simply replace `dfs(0)` with `bfs` from source `s = 0`.

Exercise 4.2.2.2: Adjacency Matrix, Adjacency List, and Edge List require $O(V)$, $O(k)$, and $O(E)$ to enumerate the list of neighbors of a vertex, respectively (note: k is the number of actual neighbors of a vertex). Since DFS and BFS explores all outgoing edges of each vertex, it's runtime depends on the underlying graph data structure speed in enumerating neighbors. Therefore, the time complexity of DFS and BFS are $O(V \times V = V^2)$, $O(max(V, V\sum_{i=0}^{V-1} k_i) = V + E)$, and $O(V \times E = VE)$ to traverse graph stored in an Adjacency Matrix, Adjacency List, and Edge List, respectively. As Adjacency List is the most efficient data structure for graph traversal, it may be beneficial to convert Adjacency Matrix or Edge List to Adjacency List first (see **Exercise 2.4.1.2***) before traversing the graph.

Exercise 4.2.3.1: Start with disjoint vertices. For each `edge(u, v)`, do `unionSet(u, v)`. The state of disjoint sets after processing all edges represent the connected components. BFS solution is 'trivial': Simply change `dfs(i)` to `bfs(i)`. Both run in $O(V + E)$.

Exercise 4.2.5.1: This is a kind of 'post-order traversal' in binary tree traversal terminology. Function `dfs2` visits all the children of u before appending vertex u at the back of vector `ts`. This satisfies the topological sort property!

Exercise 4.2.5.2: The answer is to use a Linked List. However, since in Chapter 2, we have said that we want to avoid using Linked List, we decide to use `vi ts` here.

Exercise 4.2.5.3: The algorithm will still terminate, but the output is now irrelevant as a non DAG has no topological sort.

Exercise 4.2.5.4: We must use recursive backtracking to do so.

Exercise 4.2.6.3: Proof by contradiction. Assume that a Bipartite Graph has an odd (length) cycle. Let the odd cycle contains $2k + 1$ vertices for a certain integer k that forms this path: $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_{2k-1} \rightarrow v_{2k} \rightarrow v_0$. Now, we can put v_0 in the left set, v_1 in the right set, ..., v_{2k} on the left set again, but then we have edge (v_{2k}, v_0) that is not in the left set. This is not a cycle $\rightarrow$ contradiction. Therefore, a Bipartite Graph has no odd cycle. This property can be important to solve some problems involving Bipartite Graph.

Exercise 4.2.7.1: Two back edges: $2 \rightarrow 1$ and $6 \rightarrow 4$.

Exercise 4.2.8.1: Articulation points: 1, 3 and 6; Bridges: 0-1, 3-4, 6-7, and 6-8.

Exercise 4.2.9.1: Proof by contradiction. Assume that there exists a path from vertex u to w and w to v where w is outside the SCC. From this, we can conclude that we can travel from vertex w to any vertices in the SCC and from any vertices in the SCC to w. Therefore, vertex w should be in the SCC. Contradiction. So there is no path between two vertices in an SCC that ever leaves the SCC.

Exercise 4.3.2.1: We can stop when the number of disjoint sets is already one. The simple modification: Change the start of the MST loop from: `for (int i = 0; i < E; i++) {`
To: `for (int i = 0; i < E && disjointSetSize > 1; i++) {`
Alternatively, we count the number of edges taken so far. Once it hits $V - 1$, we can stop.

Exercise 4.3.4.1: We found that MS 'Forest' and Second Best ST problems are harder to be solved with Prim's algorithm.

Exercise 4.4.2.1: For this variant, the solution is easy. Simply enqueue all the sources and set `dist[s] = 0` for all the sources before running the BFS loop. As this is just one BFS call, it runs in $O(V + E)$.

Exercise 4.4.2.2: At the start of the while loop, when we pop up the front most vertex from the queue, we check if that vertex is the destination. If it is, we break the loop there. The worst time complexity is still $O(V + E)$ but our BFS will stop sooner if the destination vertex is close to the source vertex.

Exercise 4.4.2.3: You can transform that constant-weighted graph into an unweighted graph by replacing all edge weights with ones. The SSSP information obtained by BFS is then multiplied with the constant C to get the actual answers.

Exercise 4.4.3.1: On positive weighted graph, yes. Each vertex will only be processed once. Each time a vertex is processed, we try to relax its neighbors. Because of lazy deletion, we may have at most $O(E)$ items in the priority queue at a certain time, but this is still $O(\log E) = O(\log V^2) = O(2 \times \log V) = O(\log V)$ per each dequeue or enqueue operations. Thus, the time complexity remains at $O((V + E) \log V)$. On graph with (a few) negative weight edges but no negative cycle, it runs slower due to the need of re-processing processed vertices but the shortest paths values are correct (unlike the Dijkstra's implementation shown in [7]). This is shown in an example in Section 4.4.4. On rare cases, this Dijkstra's implementation can run very slow on certain graph with some negative weight edges although the graph has no negative cycle (see **Exercise 4.4.3.2***). If the graph has negative cycle, this Dijkstra's implementation variant will be trapped in an infinite loop.

Exercise 4.4.3.3: Use `set<ii>`. This set stores sorted pair of vertex information as shown in Section 4.4.3. Vertex with the minimum distance is the first element in the (sorted) set. To update the distance of a certain vertex from source, we search and then delete the old value pair. Then we insert a new value pair. As we process each vertex and edge once and each time we access `set<ii>` in $O(\log V)$, the overall time complexity of Dijkstra's implementation variant using `set<ii>` is still $O((V + E) \log V)$.

Exercise 4.4.3.4: In Section 2.3, we have shown the way to reverse the default max heap of C++ STL `priority_queue` into a min heap by multiplying the sort keys with -1.

Exercise 4.4.3.5: Similar answer as with **Exercise 4.4.2.2** if the given weighted graph has no negative weight edge. There is a potential for wrong answer if the given weighted graph has negative weight edge.

Exercise 4.4.3.6: No, we cannot use DP. The state and transition modeling outlined in Section 4.4.3 creates a State-Space graph that is *not* a DAG. For example, we can start from state $(s, 0)$, add 1 unit of fuel at vertex s to reach state $(s, 1)$, go to a neighbor vertex y—suppose it is just 1 unit distance away—to reach state $(y, 0)$, add 1 unit of fuel again at vertex y to reach state $(y, 1)$, and then return back to state $(s, 0)$ (a cycle). So, this problem is a shortest path problem on general weighted graph. We need to use Dijkstra's algorithm.

Exercise 4.4.4.1: This is because initially only the source vertex has the correct distance information. Then, every time we relax all E edges, we guarantee that at least one more vertex with one more hop (in terms of edges used in the shortest path from source) has the correct distance information. In **Exercise 4.4.1.1**, we have seen that the shortest path must be a simple path (has at most $E = V - 1$ edges. So, after $V - 1$ pass of Bellman Ford's, even the vertex with the largest number of hops will have the correct distance information.

Exercise 4.4.4.2: Put a boolean flag `modified = false` in the outermost loop (the one that repeats all E edges relaxation $V - 1$ times). If at least one relaxation operation is done in the inner loops (the one that explores all E edges), set `modified = true`. Immediately break the outermost loop if `modified` is still false after all E edges have been examined. If this no-relaxation happens at the outermost loop iteration i, there will be no further relaxation in iteration $i + 1$, $i + 2$, ..., $i = V - 1$ either.

Exercise 4.5.1.1: This is because we will add `AdjMat[i][k] + AdjMat[k][j]` which will *overflow* if both `AdjMat[i][k]` and `AdjMat[k][j]` are near the MAX_INT range, thus giving wrong answer.

Exercise 4.5.1.2: Floyd Warshall's works in graph with negative weight edges. For graph with negative cycle, see Section 4.5.3 about 'finding negative cycle'.

Exercise 4.5.3.1: Running Warshall's algorithm directly on a graph with $V \leq 1000$ will result in TLE. Since the number of queries is low, we can afford to run $O(V + E)$ DFS per query to check if vertex u and v are connected by a path. If the input graph is directed, we can find the SCCs of the directed graphs first in $O(V + E)$. If u and v belong to the same SCC, then u will surely reach v. This can be tested with no additional cost. If SCC that contains u has a directed edge to SCC that contains v, then u will also reach v. But the connectivity check between different SCCs is much harder to check and we may as well just use a normal DFS to get the answer.

Exercise 4.5.3.3: In Floyd Warshall's, replace addition with multiplication and set the main diagonal to 1.0. After we run Floyd Warshall's, we check if the main diagonal > 1.0.

Exercise 4.6.3.1: A. 150; B = 125; C = 60.

Exercise 4.6.3.2: In the updated code below, we use *both* Adjacency List (for fast enumeration of neighbors; do not forget to include backward edges due to backward flow) and Adjacency Matrix (for fast access to residual capacity) of the same flow graph, i.e. we concentrate on improving this line: `for (int v = 0; v < MAX_V; v++)`. We also replace `vi dist(MAX_V, INF);` to `bitset<MAX_V> visited` to speed up the code a little bit more.

```
// inside int main(), assume that we have both res (AdjMatrix) and AdjList
  mf = 0;
  while (1) {                 // now a true O(VE^2) Edmonds Karp's algorithm
    f = 0;
    bitset<MAX_V> vis; vis[s] = true;       // we change vi dist to bitset!
    queue<int> q; q.push(s);
    p.assign(MAX_V, -1);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (int j = 0; j < (int)AdjList[u].size(); j++) {   // AdjList here!
        int v = AdjList[u][j];                  // we use vector<vi> AdjList
        if (res[u][v] > 0 && !vis[v])
          vis[v] = true, q.push(v), p[v] = u;
      }
    }
    augment(t, INF);
    if (f == 0) break;
    mf += f;
  }
```

Exercise 4.6.4.1: We use ∞ for the capacity of the 'middle directed edges' between the left and the right sets of the bipartite graph for the overall correctness of this flow graph modeling. If the capacities from the right set to sink t is *not* 1 as in UVa 259, we will get wrong Max Flow value if we set the capacity of these 'middle directed edges' to 1.

4.9 Chapter Notes

We end this relatively long chapter by making a remark that this chapter has lots of algorithms and algorithm inventors—the most in this book. This trend will likely increase in the future, i.e. there will be *more* graph algorithms. However, we have to warn the contestants that recent ICPCs and IOIs usually do not just ask contestants to solve problems involving the pure form of these graph algorithms. New problems usually require contestants to use creative graph modeling, combine two or more algorithms or to combine an algorithm with some advanced data structures, e.g. combining the longest path in DAG with Segment Tree data structure; using SCC contraction of Directed Graph to transform the graph into DAG before solving the actual problem on DAG; etc. These harder forms of graph problems are discussed in Section 8.4.

This chapter, albeit already quite long, still omits many known graph algorithms and graph problems that may be tested in ICPCs, namely: k-th shortest paths, Bitonic Traveling Salesman Problem (see Section 9.2), **Chu Liu Edmonds algorithm** for Min Cost Arborescence problem, **Hopcroft Karp's** MCBM algorithm (see Section 9.12), **Kuhn Munkres's (Hungarian)** weighted MCBM algorithm, **Edmonds's Matching** algorithm for general graph, etc. We invite readers to check Chapter 9 for some of these algorithms.

If you want to increase your winning chance in ACM ICPC, please spend some time to study more graph algorithms/problems beyond[27] this book. These harder ones rarely appears in *regional* contests and if they are, they usually become the *decider* problem. Harder graph problems are more likely to appear in the ACM ICPC World Finals level.

However, we have good news for IOI contestants. We believe that most graph materials in the IOI syllabus are already covered in this chapter. You need to master the basic algorithms covered in this chapter and then improve your problem solving skills in applying these basic algorithms to creative graph problems frequently posed in IOI.

Statistics	First Edition	Second Edition	Third Edition
Number of Pages	35	49 (+40%)	70 (+43%)
Written Exercises	8	30 (+275%)	30+20*=50 (+63%)
Programming Exercises	173	230 (+33%)	248 (+8%)

The breakdown of the number of programming exercises from each section is shown below:

Section	Title	Appearance	% in Chapter	% in Book
4.2	**Graph Traversal**	65	26%	4%
4.10	Minimum Spanning Tree	25	10%	1%
4.4	Single-Source Shortest Paths	51	21%	3%
4.5	All-Pairs Shortest Paths	27	11%	2%
4.6	Network Flow	13	5%	1%
4.7	**Special Graphs**	67	27%	4%

[27] Interested readers are welcome to explore Felix's paper [23] that discusses maximum flow algorithm for *large* graphs of 411 million vertices and 31 billion edges!