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.

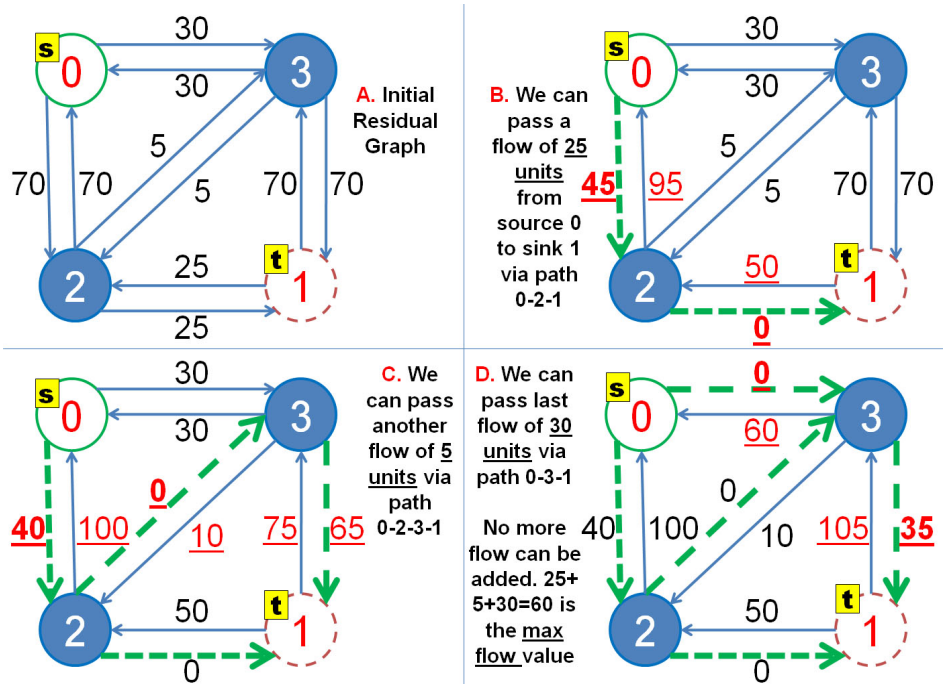

Figure 4.24: Max Flow Illustration (UVa 820 [47] - ICPC World Finals 2000 Problem E)

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.

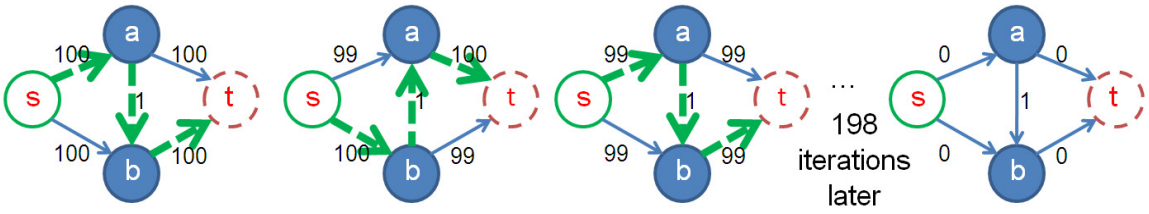

Figure 4.25: Ford Fulkerson's Method Implemented with DFS Can Be Slow

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!

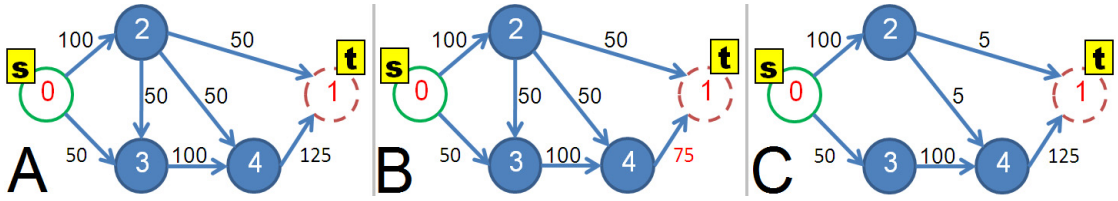

Figure 4.26: What are the Max Flow value of these three residual graphs?

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.

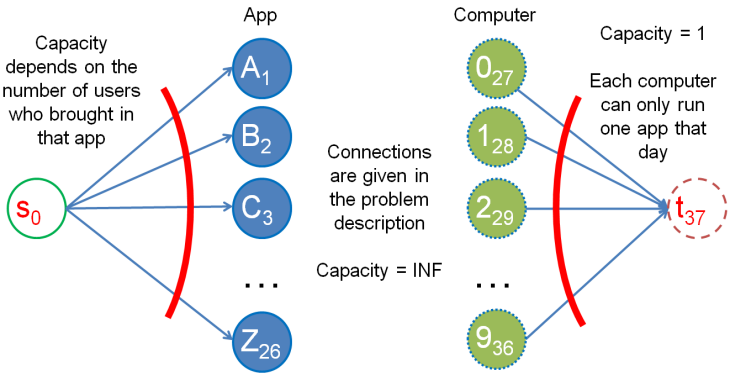

Figure 4.27: Residual Graph of UVa 259 [47]

[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

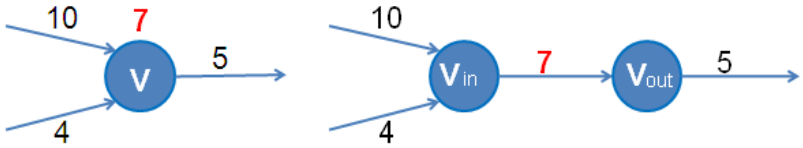

Figure 4.28: Vertex Splitting Technique

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.

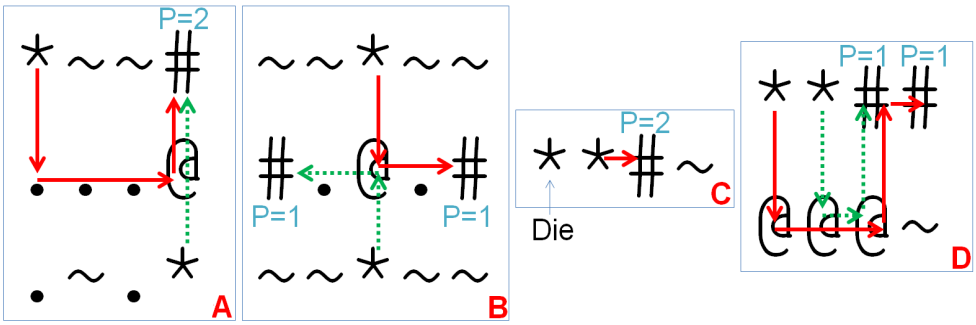

Figure 4.29: Some Test Cases of UVa 11380

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.

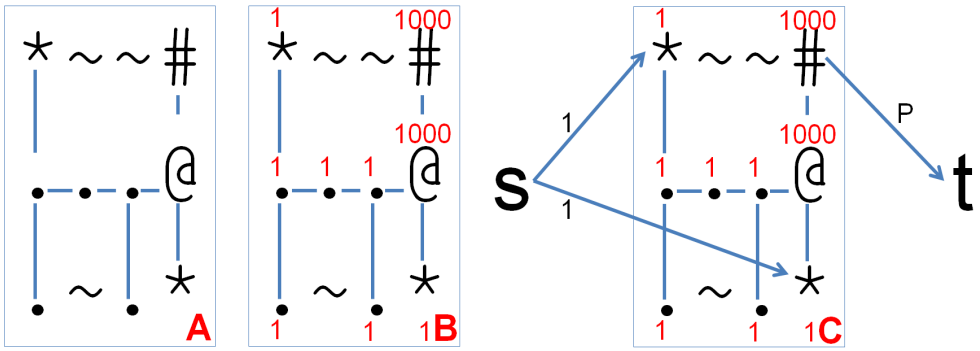

Figure 4.30: Flow Graph Modeling

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].

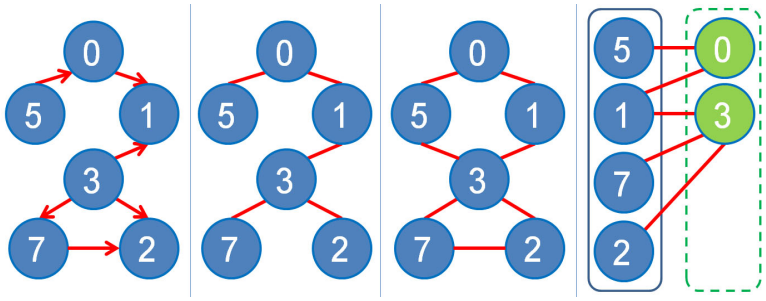

Figure 4.31: Special Graphs (L-to-R): DAG, Tree, Eulerian, Bipartite Graph

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.

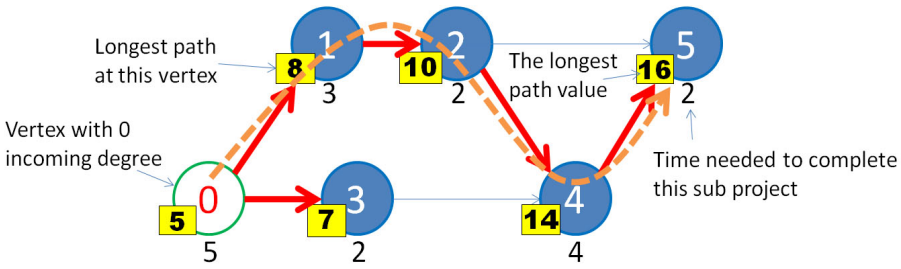

Figure 4.32: The Longest Path on this DAG

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).

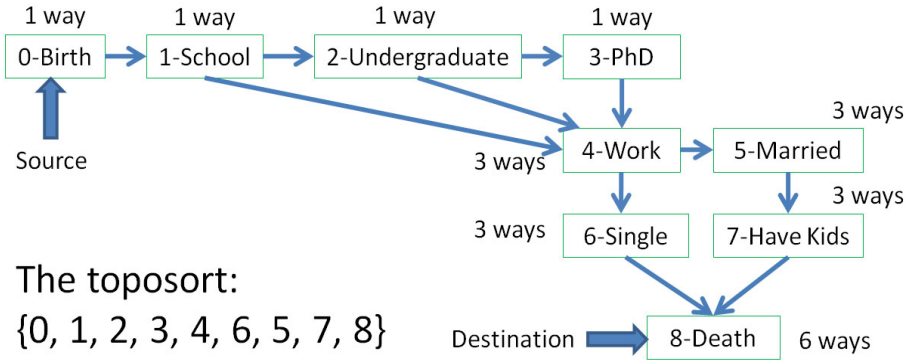

Figure 4.33: Example of Counting Paths in DAG - Bottom-Up

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.

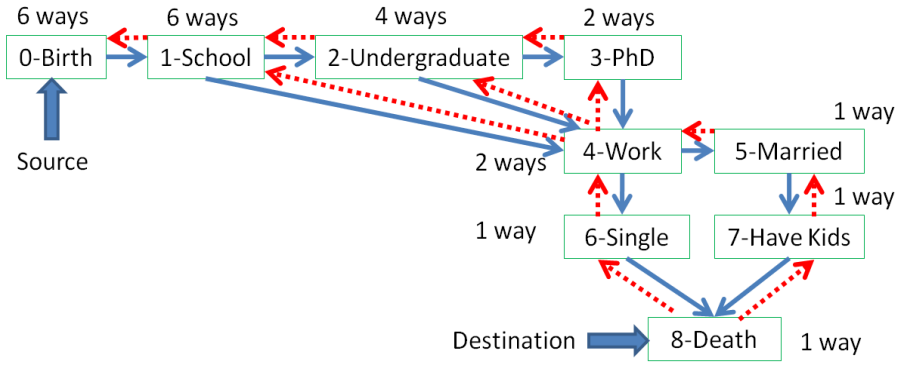

Figure 4.34: Example of Counting Paths in DAG - Top-Down

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.

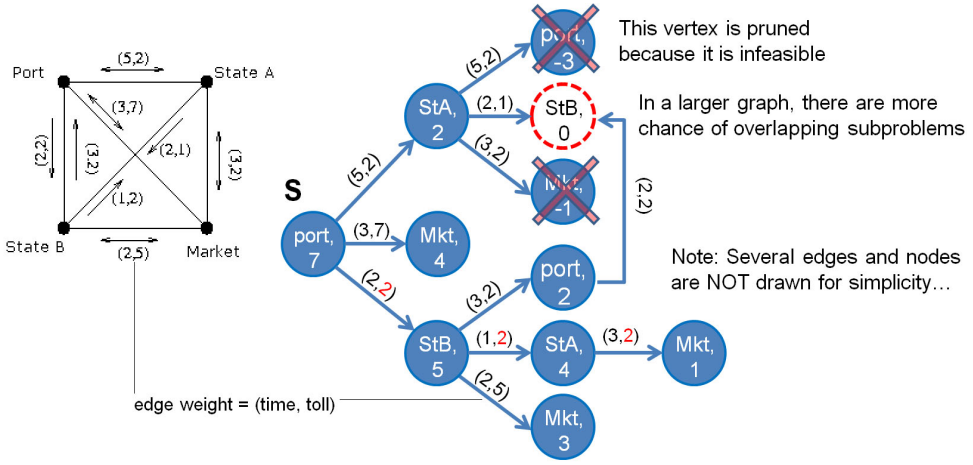

Figure 4.35: The Given General Graph (left) is Converted to DAG

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.

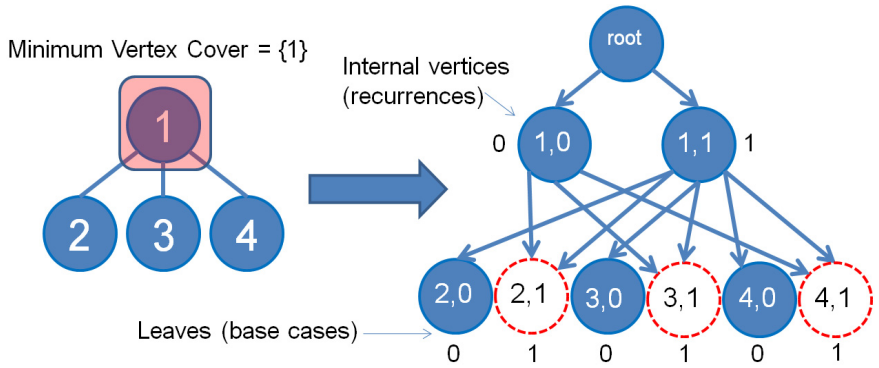

Figure 4.36: The Given General Graph/Tree (left) is Converted to DAG

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: $10 \rightarrow 5 \rightarrow 0$.

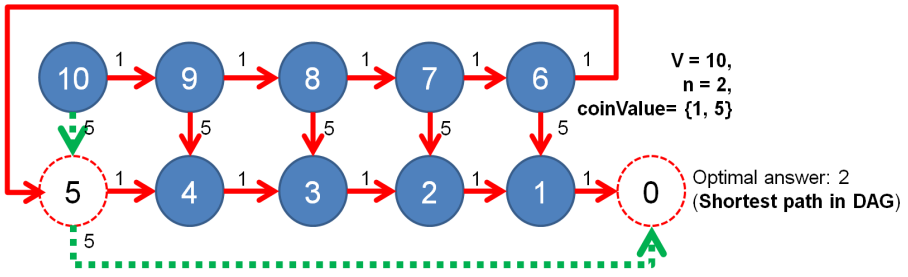

Figure 4.37: Coin Change as Shortest Paths on DAG