NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

SC1007 Data Structures and Algorithms **2022/23 Semester 2**

11: Backtracking, Permutation and Matching

School of Computer Science and Engineering *Nanyang Technological University*

A major challenge is how to define the solution space of a problem. We need to translate a real-life problem and its all possible solutions into a graph problem. Then we can design an algorithm to find optimal solutions from the graph. The solution space should contain at least one optimal solution, so that we can search it in the space. This is challenging as we need to understand the problem itself, and then rephrase or model it into a graph problem.

In this lecture, we will learn two applications using graph. An application of depth first search in artificial intelligence; it is called **Backtracking**. Another application is matching problem using bipartite graph.

11.1 Backtracking

Backtracking is a strategy of algorithm design for searching solutions. It is actually an application of Depth First Search (DFS) which you have learned in previous lecture.

The first node to search is the initial state of the problem. Here the state means the configuration of a solution. An example is that while playing chess, the initial state is that there is no chess piece on the chess board.

From the starting node, we will search in a depth-first manner in the solution space. For example, when we analyze the running time of a recursive algorithm, we follow a sequence of steps. Firstly, we derive a recurrence equation; secondly we solve the equation by repeated substitutions; and lastly, we use the big O notation to represent the time complexity. Going through these steps is like moving down the path of a tree. However, our experiences tell us that it is not always that smooth. Sometimes we are just stuck at some step of derivation, maybe due to some mistake in a previous step. In that case, we need to do backtracking. When we encounter a dead end during the search, it means we cannot proceed any further, or it is impossible to find a correct solution along the path. Thus, we have to backtrack to the most recent node and try an alternative path.

When should we stop the search? The search will terminate when we either have found the optimal solution, or once we have run out of nodes to backtrack to. The first case applies when finding one solution is enough. It does not pose as a concern if there are other possible solutions. The latter is when we need to find all possible solutions, and after all possible paths have been explored, we return to the starting node. Thus, the terminating conditions will depend on the problem.

11.2 The Map Coloring Problem

The map coloring problem is a classic problem in graph theory, which involves assigning colors to the regions of a map so that no two adjacent regions have the same color. In other words, the goal is to color a map with the minimum number of colors such that no two adjacent regions have the same color.

One of the earliest known references to the problem appears in the work of Francis Guthrie, a mathematics student at University College London in the mid-19th century. The story goes that Guthrie was trying to color the counties of England on a map, and he noticed that it was always possible to color the map with only four colors so that no adjacent regions had the same color. He then posed the question to his brother, who was also a mathematician, and the problem eventually made its way to Augustus De Morgan, a famous mathematician of the time. De Morgan discussed the problem with his colleague William Hamilton, and the problem became known as the "Four Color Problem". In the 20th century, the problem gained more attention from mathematicians and computer scientists, and in 1976, Kenneth Appel and Wolfgang Haken at the University of Illinois at Urbana-Champaign finally proved that any map can be colored with at most four colors. Their proof was based on an extensive computer-based analysis of many different cases, and it was controversial at the time because of the heavy use of computer algorithms in the proof. However, subsequent research has confirmed the validity of their proof, and the Four Color Problem is now considered to be one of the most famous and important problems in graph theory.

This map coloring problem can be solved by a backtracking algorithm. Here's a general outline of how a backtracking algorithm could be used to solve the map coloring problem:

1. Start with an initial coloring of the map, such as assigning the first color to the first region.
2. Select an uncolored region and try to assign it a color.
3. Check if the color assignment is valid, i.e., check if no adjacent regions have the same color.
4. If the color assignment is valid, move to the next uncolored region and repeat steps 2-3.
5. If the color assignment is not valid, backtrack to the previous region and try a different color assignment for that region.
6. If all possible color assignments for a region have been tried and none of them are valid, backtrack to the previous region and try a different color assignment for that region.
7. Repeat steps 2-6 until all regions have been colored or until it is determined that a valid coloring cannot be found.

The backtracking algorithm works by recursively trying different color assignments and backtracking when a coloring is found to be invalid. The algorithm explores all possible colorings until a valid coloring is found or all possibilities have been exhausted.

However, note that the backtracking algorithm for the map coloring problem can be very computationally expensive, especially for large maps. Therefore, other algorithms and heuristics, such as greedy algorithms and constraint satisfaction algorithms, are often used to solve the problem more efficiently.

11.3 The Eight Queen Problem

The eight queens problem is one of the classical computer science problem. The problem was first published in 1848 by Max Bezzel. In 1972, Prof. Edsger Dijkstra who introduced Dijkstra's shortest path algorithm use DFS algorithm to solve the problem.

The problem is to placing eight chess queen on an 8×8 chessboard so that no two queens can attach each other. Since the given chessboard has 8 rows and 8 columns, there are 64 possible grids to place 8 queens. If we use exhausting search to find the solution, there are $\binom{64}{8} = 4.43$ billion ways.

If we know that each row can only place a queen, the number of possible solutions will be reduced to $8! = 40320$. Can we do it better?

As we know, a queen can move within its row or its column or along its diagonal.

The main idea is that we are going to dynamically generate all possible solutions to the problem as a tree. We will systematically search for a correct or optimal solution from the tree.

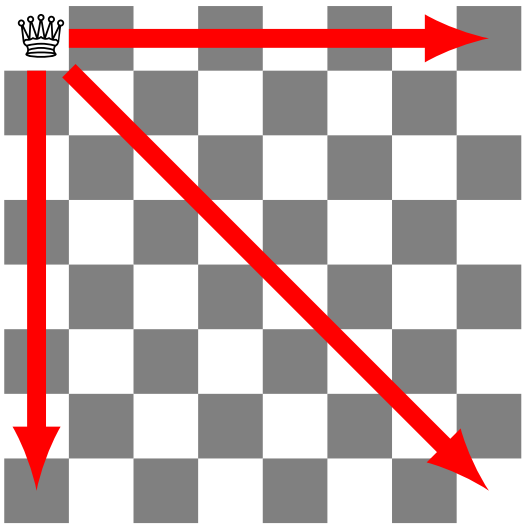


Figure 11.1: A queen can move within its row or its column or along its diagonal

11.3.1 Backtracking Algorithm

1. Starts by placing a queen in the top left corner of the chess board.

2. Places a queen in the second column and moves her until a place where she cannot be hit by the queen in the first column.

3. Places a queen in the third column and moves her until she cannot be hit by either of the first two queens and so on.

4. If there is no place for the i^{th} queen, the program backtracks to move the $(i-1)^{th}$ queen.

5. If the $(i-1)^{th}$ queen is at the end of the column, the program removes the queen and backtracks to the $(i-2)$ column and so on.

6. If the current column is the last column and a safe place has been found for the last queen, then a solution to the puzzle has been found.

If the current column is the first column and its queen is being moved off the board then all possible configurations have been examined, all solutions have been found, and the algorithm terminates.

The following is the pseudo code of finding a solution of the N queens problem:

Algorithm 1 Backing Tracking Algorithm for The Eight Queens Problem

```
function NQueens(Board[N][N], Column)
    if Column >= N then return true                          ▷ Solution is found
    else
        for i ← 1, N do
            if Board[i][Column] is safe to place then
                Place a queen in the square
                if NQueens(Board[N][N], Column + 1) then return true   ▷ Solution is found
                end if
                Delete the queen
            end if
        end for
    end if
    return false                                             ▷ no solution is found
    end function
```

11.3.2 Finding solution for the Four Queens Problem

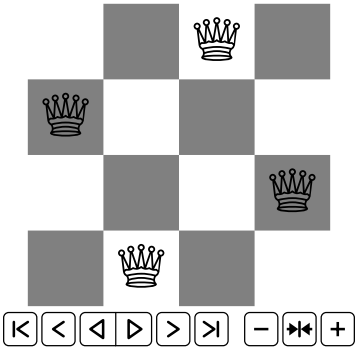
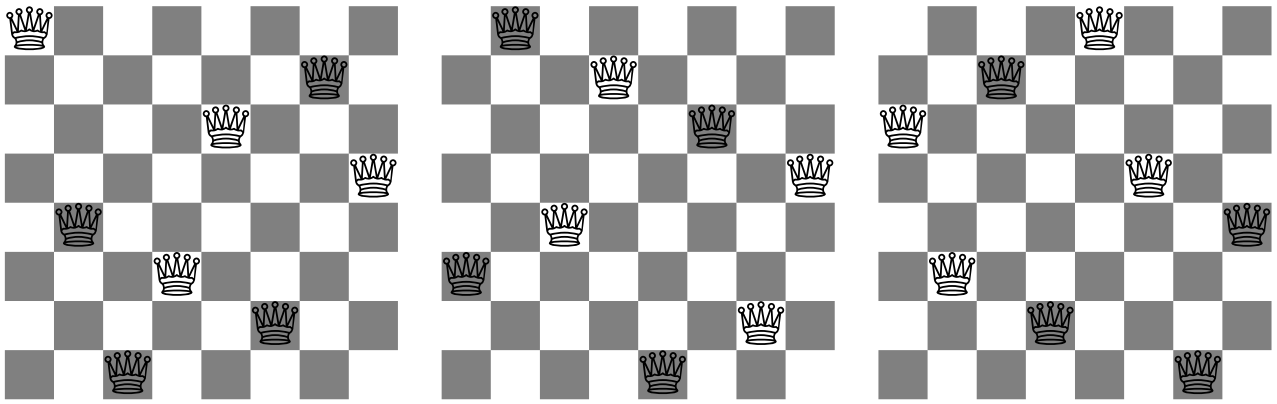


Figure 11.2: Four Queens Problem

11.3.3 Some Solutions



(a) Solution 1 (b) Solution 2 (c) Solution 3

Figure 11.3: Three solutions of the eight queens problem

It is noted that this puzzle has 92 solutions.

11.4 Dynamic Programming

Dynamic Programming (DP) was introduced by Richard Bellman in 1953. It is not a programming language like C, Python and Java. "Programming" here refers to a planning method. It is similar to divide-and-conquer method, which solves the problems by combining the solutions of its subproblems. However, the subproblems can be identical. If it is the case, divide-and-conquer method will repeatedly solve the same subproblem. It is very inefficient. To improve the efficiency, DP uses tables to "remember" the solutions of the subproblems. Thus, the solutions can be found easily when you encounter the same subproblem again. In computing, it is known as **memoization** which was coined by Donald Michie in 1968 and is derived from the Latin word "memorandum".

DP is typically applied to optimization problems. Other "programming" methods in mathematical optimization are linear programming, integer programming, convex programming and semidefinite programming. These methods are trying to find the optimal solutions from many possible solutions of an optimization problem. It has been widely applied from system control to economics.

When we are developing a DP algorithm, we first construct the optimal substructure and its solution. The optimal solution usually can be obtained recursively. If we simply use divide-and-conquer strategy to solve the problem recursively, its time complexity will be in $\Theta(2^n)$. However, many subproblems are repeatedly computed. Thus, if we use memoization approach to store the optimal solutions to sub-problems in a table (or memory or cache), the time complexity can be improved to polynomial class, $\Theta(n^p)$.

There are two way to realise a DP approach. The first approach is the top-down approach with memoization. It modified the recursive solution by saving the result of each subproblem in an array or a hash table. Instead of recompute the same sub-problem, this approach always check the array or table if the solution is stored. It serves an example of a time-space trade-off.

The second approach is the bottom-up method. Since any particular subproblem depend only on solving "smaller" subproblems, we sort the subproblems by size and solve them in size order, smallest first. Thus,

we ensure that each subproblem is only solved once.

11.4.1 Example 1: Fibonacci Number

Although DP uses tables to store solution of subproblems, you may interpreted as a special variety of space-for-time trade-off, DP can sometimes be refined to avoid using extra space. In this example (see the algorithms in Section 1.3.2), we only need to store the last two elements of the Fibonacci sequence for the next term. It is noted that DP can be applied only when the sub-problems are not independent.

11.4.2 Example 2: Rod-Cutting Problem

Given a rod of a certain length and price of rod of different lengths, determine the maximum revenue obtainable by cutting up the rod at different lengths based on the prices.

For example, the price of rods of different length is as given in the table below:

Length	1	2	3	4	5	6	7	8	9	10
Price SGD	1	5	8	9	10	17	17	20	24	30

If we have a rod of length 4, the price of it is SGD9 if we do not cut it into pieces. However, we can sell SGD5 each if we cut the rod into two pieces of length 2 each. The total revenue will increase to SGD10.

The challenge is to find a systematic and efficient approach to obtain the maximum revenue. In this example, all possible ways of cutting the rod and their total revenue are showed in the table below:

Length of each piece	Total revenue
4	9
1,3	9
1,1,2	7
1,1,1,1	4
2,2	**10**

The naïve recursive approach to

Algorithm 2 Rod-Cutting Problem: Recursive Approach

```
1: function CUT-ROD(p, n)
2:     if n == 0 then
3:         return 0
4:     end if
5:     q ← −∞
6:     for i ← 1, . . . , n do
7:         q ←max(q,p[i]+Cut-Rod(p, n − i))
8:     end for
9:     return q
10: end function
```

In every recursive call, cut the rod into two pieces of length i and n-i. The second piece will be further cut into two piece via another recursive call to find its max revenue. The recursive calls will repeatedly find the revenue for a rod of the same length. Its time complexity is $\Theta(2^n)$.

In memoization, we store the optimal solution of each sub-problem when it is first computed and then reused when the same subproblem is encountered again.

Algorithm 3 Rod-Cutting Problem: Top-down Approach with Memoization

```
1: function CUT-ROD(p, n)
2:     r[1, ..., n] ← 0
3:     return Mem-Cut-Rod-Aux (p, n, r)
4: end function
5: function MEM-CUT-ROD-AUX(p,n,r)
6:     if n == 0 then
7:         return 0
8:     end if
9:     if r[n] > 0 then
10:        return r[n]
11:    else
12:        q ← −∞
13:        for i ← 1, ..., n do
14:            q ←max(q, p[i]+Mem-Cut-Rod-Aux(p, n − i, r))
15:        end for
16:        r[n] ← q
17:    end if
18:    return q
19: end function
```

In some circumstances, the top-down approach does not need to evaluate all possible sub-problems but it has overhead for recursive calls. The bottom-up approach which has the same asymptotic running time but less overhead for procedure calls can be considered.

Algorithm 4 Rod-Cutting Problem: Bottom-Up Approach

```
1: function DP-CUT-ROD(p, n)
2:     r[1, ..., n] ← 0
3:     for j ← 1, ..., n do
4:         for i ← 1, ..., j do
5:             r[j] ←max(r[j], p[i] + r[j − i])
6:         end for
7:     end for
8:     return r[n]
9: end function
```

In the earlier example, the maximum revenue of each rod of different can be found in the table below

Length	1	2	3	4	5	6	7	8	9	10
Max Rev.	1	5	8	10	13	17	18	22	25	30

11.4.3 Example 3: 0/1 Knapsack Problem

Given n items of known weights $s_1, \ldots, s_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity C, we would like to find the largest total value of the subset of the items that fit in the knapsack. We assume that all the weights and the knapsack capacity are positive integer but their value do not have to be integers.

This problem can be written as a optimization problem

$$\begin{aligned} \max_{x} \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} s_i x_i \leq C \\ & x_i \in \{0,1\} \qquad\qquad i = 1, 2, \ldots, n \end{aligned} \tag{11.1}$$

The brute force algorithm is to consider all possible subsets of items and compute their total weights and values. For those subsets with total weight lesser than capacity C, we find the subset of the maxumum value. The i^{th} item is either included (1) or excluded (0). Three items will have $2^3 = 8$ possible subsets.

item 1	item 2	item 3	value	weight
0	0	0	0	0
0	0	1	V3	s3
0	1	0	V2	s2
0	1	1	V2+V3	s2+s3
1	0	0	V1	s1
1	0	1	V1+V3	s1+s3
1	1	0	V1+V2	s1+s2
1	1	1	V1+V2+V3	s1+s2+s3

From the example above, you may observe that some subproblems are repeatedly used. To design a DP algorithm for this problem, we need to derive a recurrence relation or its optimal substructure. Let us consider the first i items, $1 \leq i \leq n$, with weights $s_1, \ldots, s_i$ and values $v_1, \ldots, vi$ and knapsack capacity j,$1 \leq j \leq C$. Let the optimal solution be $M(i, j)$.

We need to consider two possible solutions, the i^{th} item is included and excluded.

1. Among the subsets that exclude the i^{th} item, the optimal solution is defined $M(i-1, j)$
2. Among the subsets that include the i^{th} item, the optimal solution is the value of the i^{th} and an optimal solution of the $i-1$ items that fits into the knapsack of capacity $j - s_i$. Hence, it is $v_i + M(i-1, j-s_i)$

The recurrence relation can be formulated as

$$M(i,j) = \begin{cases} \max(M(i-1,j), v_i + M(i-1, j-s_i)), & \text{if } j - s_i \geq 0. \\ M(i-1,j), & \text{otherwise.} \end{cases} \tag{11.2}$$

if the s_i exceeds the capacity j, we definitely do not consider the i^{th} item. It cannot fit into the knapsack.

Let us consider the following example which a knapsack with capacity C=5 and the weight and value of each item are given below

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

we can use bottom up DP approach with the recurrence relation to fill up the table below and find the optimal solution

	j				
i	1	2	3	4	5
1	\$0	\$12	\$12	\$12	\$12
2	\$10	\$12	\$22	\$22	\$22
3	\$10	\$12	\$22	\$30	\$32
4	\$10	\$15	\$25	\$30	**\$37**

The optimal solution is 37 (item 1, item 2 and item 4). The table is n-by-C. The time and space complexity are both in $\Theta(nC)$.

11.5 Matching Problem

Suppose that a company requires a number of different types of jobs to its workers. Each worker can do some of them but not others. Moreover, each of them can only assign one job at a time. How should the jobs be assigned so that the maximum number of jobs can be taken? This is a typical matching problem in our daily life.

If we formulate this problem into a graph problem, jobs and workers are two disjoint sets, J and W such that every edge connects a vertex in J to one in W. This graph is known as **Bipartite Graph**.

Definition 11.1 *A **matching** of a graph G is a subset of edges of G that are mutually non-adjacent. Thus, no two edges in the subset have an endpoint in common.*

Definition 11.2 *A **maximum matching** of a graph G is a matching with the maximum number of edges.*

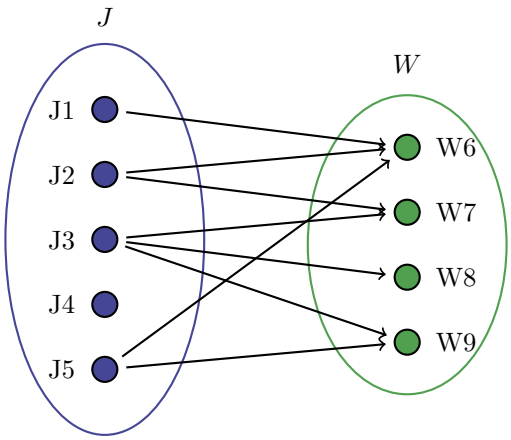


Figure 11.4: Bipartite graph for matching problem

11.5.1 Maximum Flow

We need to find as many matching edges as possible. This problem can be resolved by finding the maximum flow of the network.

Definition 11.3 *A **flow network** $G = (V, E)$ is a directed graph in which each $(j, w) \in E$ has a nonnegative capacity $c(j, w) \geq 0$.*

In the matching problem above, the capacity of each edge is 1. If $(j, w) \notin E$, then we assume that $c(j, w) = 0$. This approach can also apply on weighted matching problem. Let us discuss about unweighted graph first.

Here, we introduce two vertices into the flow network: a *source* s and a *sink* t. Now the matching problem becomes finding as many paths as we can from s to t. See Figure 11.5.

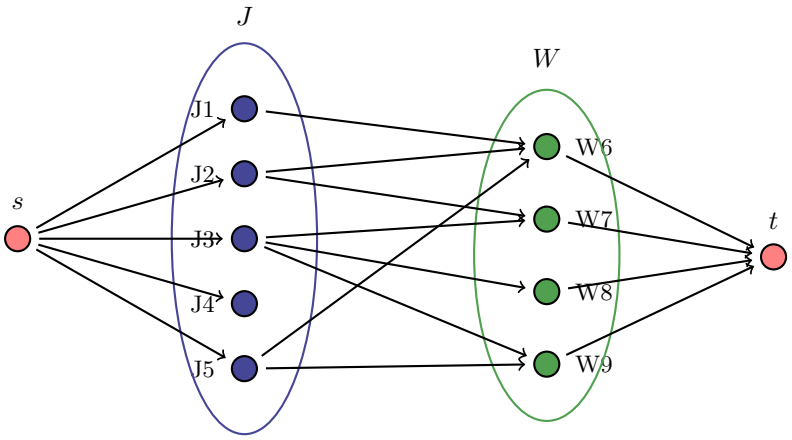


Figure 11.5: Bipartite graph for matching problem with source node and sink nodes

11.5.2 The Ford-Fulkerson Method

The Ford-Fulkerson method is using *iterative improvement* strategy to find the maximum flow in a flow network. It was proposed by L. R Ford Jr. and D. R Fulkerson in 1956.

To iteratively find the additional flow (match) in the network, we need a **residual network** which consists of edges that can admit more net flow.

Let $f(j, w)$ denote a flow at edge (j, w) in G and $c(j, w)$ denote its capacity. At a pair of vertices, j and w, the residual capacity at their edge is

$$c_f(j, w) = c(j, w) - f(j, w) \tag{11.3}$$

Given a flow network $G = (V, E)$ and a flow f, the residual network of G induced by f is $G_f = (V, E_f)$ where

$$E_f = \{e_f = (j, w) \in V \times V : c_f(j, w) > 0\} \tag{11.4}$$

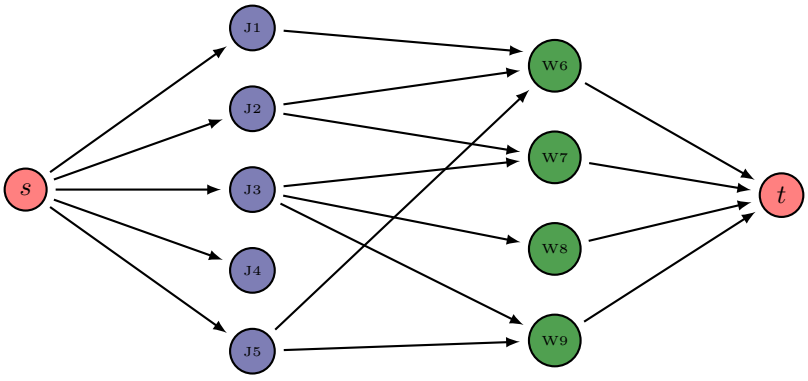
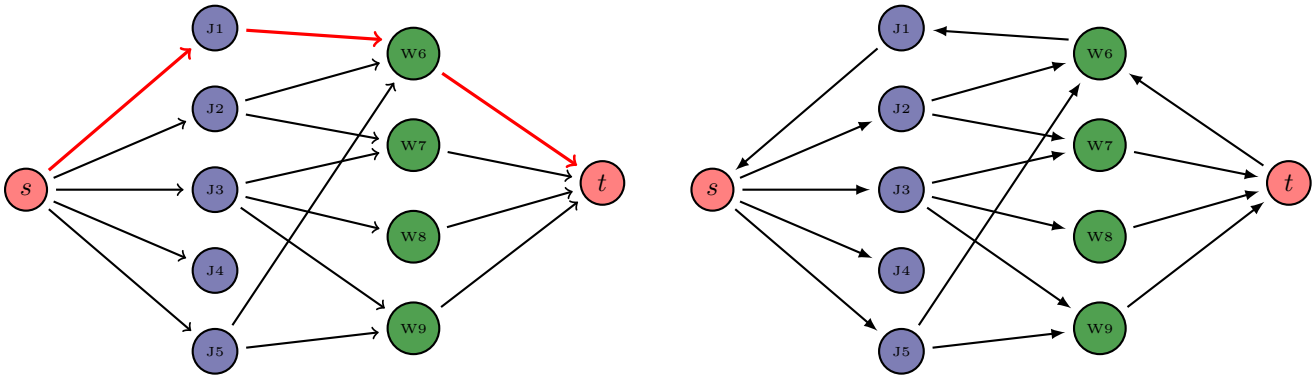


Figure 11.6: Residual Network G_f where $c_f(j, w)$ is 1 $\forall e_f$

The residual network in Figure 11.6 is the same as the original network graph. After flows are introduced to the graph, it may consists of edge $e_f = (w, j) \notin E$. Such an edge appears in G_f but not in the original flow network G only if $(j, w) \in E$ and there is positive net flow from j to w in G.

In each iteration, the Ford-Fulkerson method find an **augmenting path** p from s to t in the residual network G_f.



(a) Matching Network G (b) Residual Network G_f

Figure 11.7: After adding an augmenting path from $s \to J_1 \to W_6 \to t$, some edges in G_f are removed if their capacity is zero.

11.5.3 Algorithm of Ford-Fulkerson

Algorithm 5 Ford-Fulkerson

```
function FORD-FULKERSON(Graph G, Vertex s, Vertex t)
    for each edge (u, v) ∈ E[G] do                          ▷ Initialization of Flows
        f[u, v] ← 0
        f[v, u] ← 0
    end for
    while Finding a path from s to t in G_f do
        c_min(p) ← min{c_f(u, v) : (u, v) ∈ p}
        for each edge (u, v) ∈ p do
            if (u, v) ∈ E then
                f[u, v] ← f[u, v] + c_min(p)                ▷ adding the new flow
            else
                f[v, u] ← f[v, u] − c_min(p)
            end if
            c_f(u, v) ← c_f(u, v) − c_min(p)                ▷ Update Residual Graph, G_f
            c_f(v, u) ← c_f(v, u) + c_min(p)
        end for
    end while
end function
```

For the matching problem above, the capacity is either zero or one. If the edge with zero capacity, the edge will be ignored here. if edge (j, w) is included into a flow path, the $c_f(j, w)$ will be zero. At the same time, $c_f(w, j)$ will be added to 1. Thus, edge (j,w) is removed from residual graph G_f and a new edge (w,j) is added.

11.5.4 Using Ford-Fulkerson algorithm to solve the matching problem

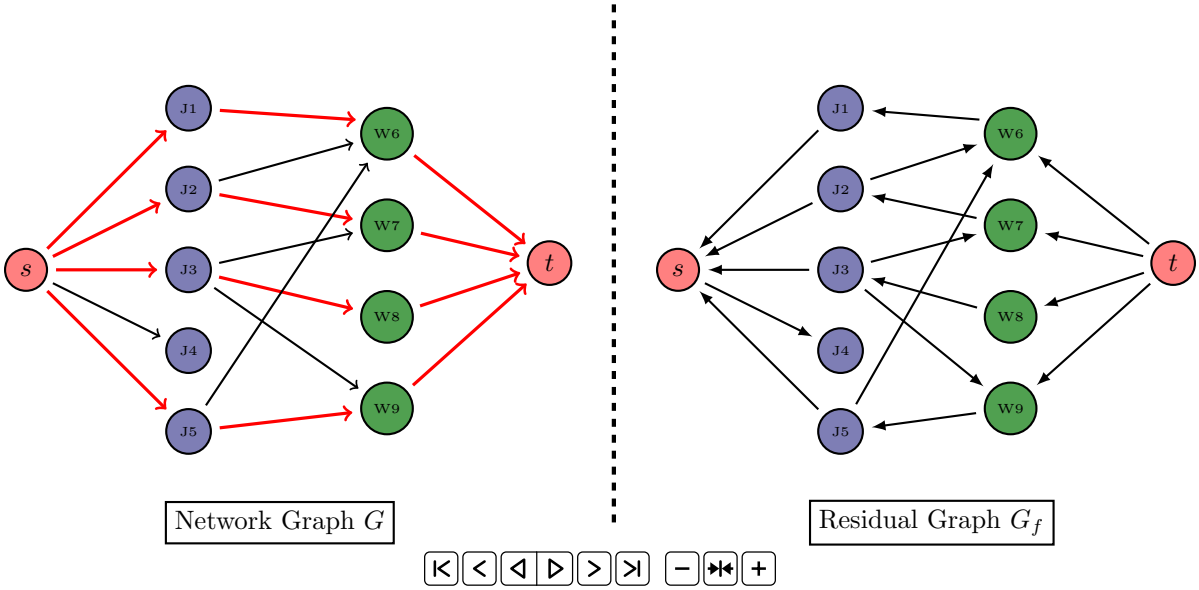


Figure 11.8: Matching Network G

The maximum matching is found when there is no path from s to t in the residual graph G_f.

11.5.5 An example of weighted graph

The same algorithm works not only on unweighed graphs, but also on weighted graphs.

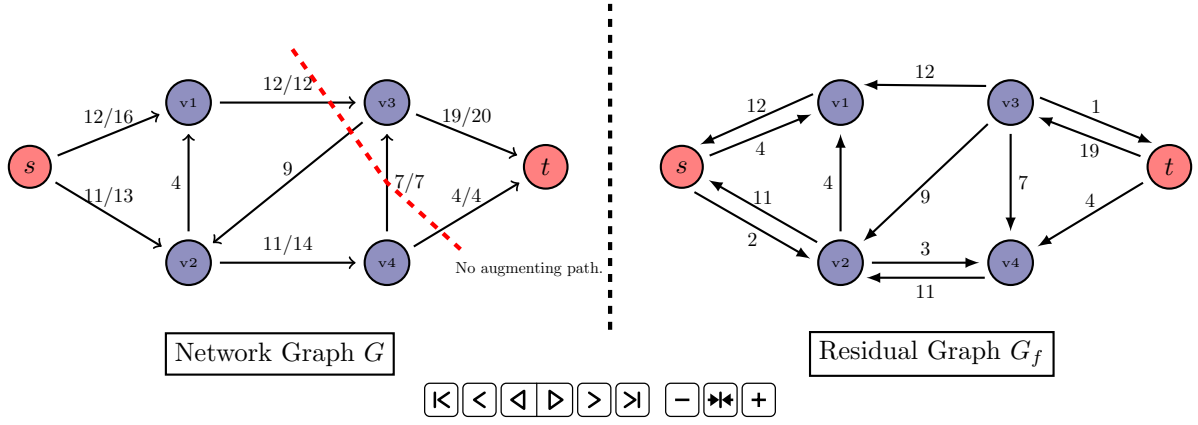


Figure 11.9: A Network Flow Problem. The value of the maximum flow is 23

In the final slide, the red dashed line $((v_1, v_3) - (v_4, v_3) - (v_4, t))$ separates the vertices into two disjoint subsets. The total weight on these edges equals to the maximum flow (23). It is also known as **minimum cut**.

11.5.6 Implementation of Ford Fulkerson Method

You may observe that finding the augmenting path is one of the key issues. If we can search the path faster and better, we will make the Ford-Fulkerson method reaching the maximum flow in fewer iterations. Generally, we can use BFS and DFS to find each augmenting path. The BFS version is also known as *Edmonds-Karp algorithm* proposed by Jack Edmonds and Richard Karp in 1972.

11.5.7 Time Complexity

Based on Ford-Fulkerson method, several improved algorithms were proposed. Their time complexity is improved to $\mathcal{O}(|V|^3)$).

Year	Proposers	Algorithms	Complexity
1951	Dantzig	Simplex	$\mathcal{O}(\|E\|\|V\|^2C)$
1955	Ford, Fulkerson	Augmenting Path	$\mathcal{O}(\|E\|\|V\|C)$
1970	Edmonds-Karp	Shortest Path	$\mathcal{O}(\|E\|^2\|V\|)$
1970	Edmonds-Karp	Max capacity	$\mathcal{O}(\|E\|\log C(\|E\|+\|V\|\log V))$
1970	Dinitz	Improved Shortest Path	$\mathcal{O}(\|E\|\|V\|^2))$
1972	Edmonds-Karp, Dinitz	Capacity Scaling	$\mathcal{O}(\|E\|^2\log C))$
1973	Dinitz-Gabow	Improved capacity scaling	$\mathcal{O}(\|E\|\|V\|\log C))$
1974	Karzanov	Preflow-push [K74]	$\mathcal{O}(\|V\|^3))$
1981	Sleator-Tarjan	Dynamic trees[S81]	$\mathcal{O}(\|E\|\|V\|\log V))$
1986	Goldberg-Tarjan	FIFO preflow-push[G86]	$\mathcal{O}(\|E\|\|V\|\log(\|V\|^2/\|E\|)))$
1998	Goldberg-Rao	Length Function [G98]	$\mathcal{O}(\min(\|V\|^{2/3},\|E\|^{1/2})\|E\|log(\|V\|^2/\|E\|)\log C)$

where C is the maximum capacity in the network. Here you do not need to derive the time complexities. It is just for your information. If you would like to know the derivation of their time complexity, please read the Cormen's *Introduction To Algorithms*.

Maximum flow and minimum-cut are widely applicable in many areas eg. image processing, network optimization, distributed computing, operations research etc.

References

[K74] A. V. Karzanov "Determining the maximal flow in a network by the method of preflows" *Soviet Math. Dokl. 15, 434-437*, 1974.

[S81] D.D. Sleator and R.E. Tarjan "A data structure for dynamic trees" *In Proc. Thirteenth Annual ACM Symp. on Theory of Computing, 114-122*, 1981.

[G86] Andrew V. Goldberg and R. E. Tarjan "A new approach to the maximum flow problem" *Proceedings of the eighteenth annual ACM symposium on Theory of computing – STOC*, 1986.

[G98] Andrew V. Goldberg and Satish Rao "Beyond the flow decomposition barrier." *J. ACM 45, 5, (Sept. 1998), 783–797*, 1998.