CX1107 Data Structures and Algorithms **2020/21 Semester 2**

12: 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.

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.

12.1 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: First, 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.

12.2 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.

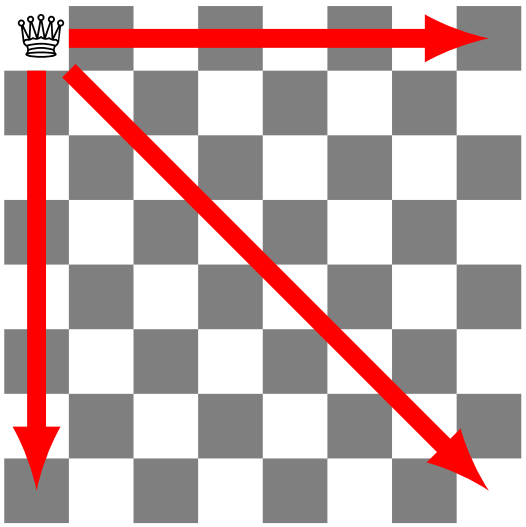


Figure 12.1: A queen can move within its row or its column or along its diagonal

12.2.1 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
```

12.2.2 Finding solution for the Four Queens Problem

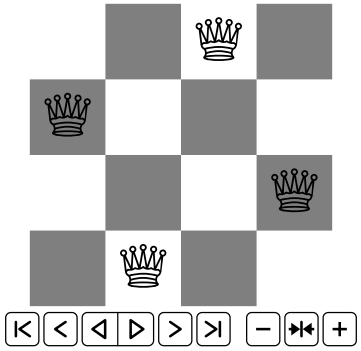
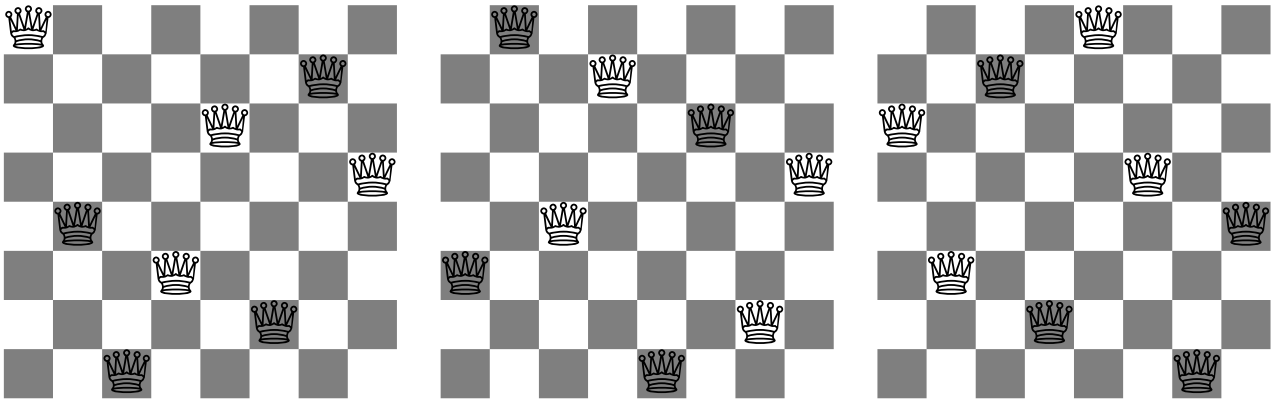


Figure 12.2: Four Queens Problem

12.2.3 Some Solutions



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

Figure 12.3: Three solutions of the eight queens problem

It is noted that this puzzle has 92 solutions.

12.3 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 12.1 *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 12.2 *A **maximum matching** of a graph G is a matching with the maximum number of edges.*

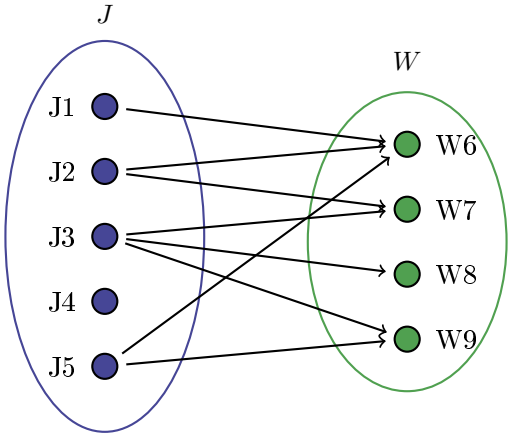


Figure 12.4: Bipartite graph for matching problem

12.3.1 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 12.3 *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 12.5.

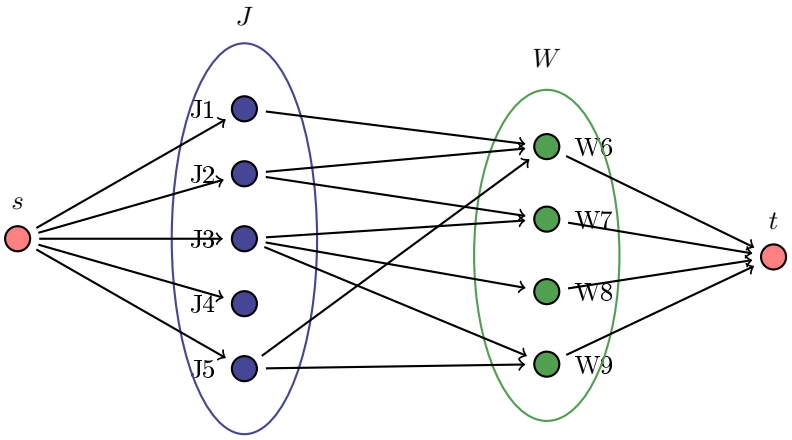


Figure 12.5: Bipartite graph for matching problem with source node and sink nodes

12.3.2 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

$$c_f(j, w) = c(j, w) - f(j, w) \tag{12.1}$$

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

$$E_f = \{e_f = (j, w) \in V \times V : c_f(j, w) > 0\} \tag{12.2}$$

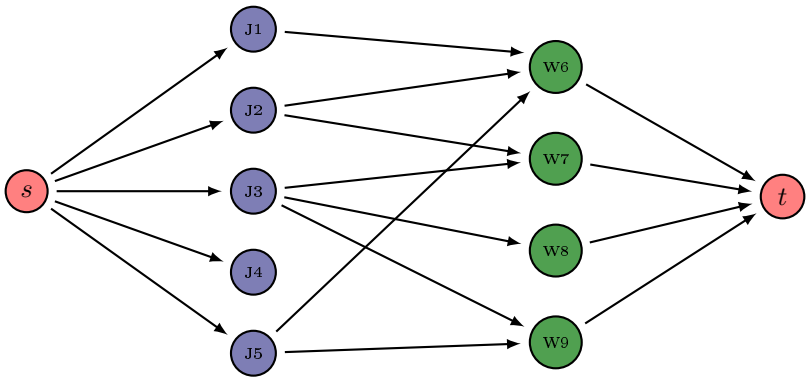
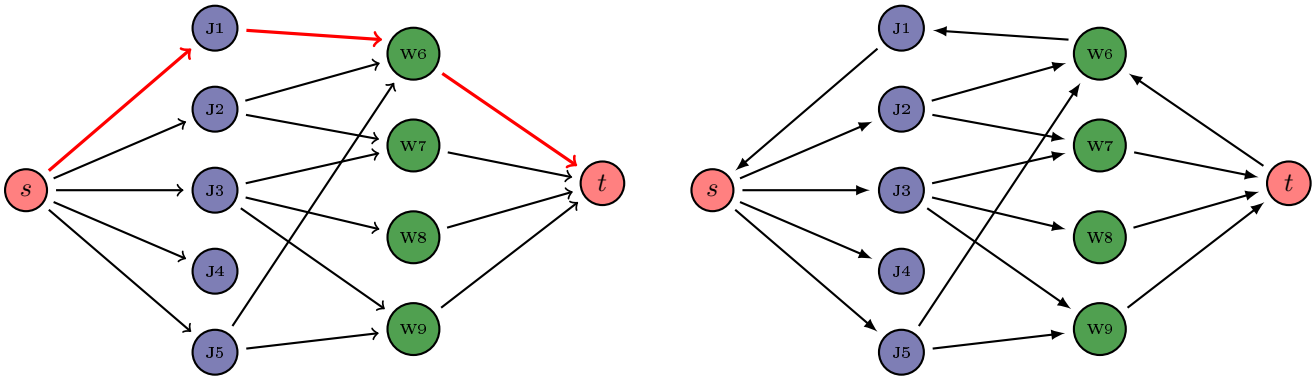


Figure 12.6: Residual Network G_f where $c_f(j, w)$ is 1 $\forall e_f$

The residual network in Figure 12.6 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 12.7: 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.

12.3.3 Algorithm of Ford-Fulkerson

Algorithm 2 Ford-Fulkerson

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

12.3.4 Using Ford-Fulkerson algorithm to solve the matching problem

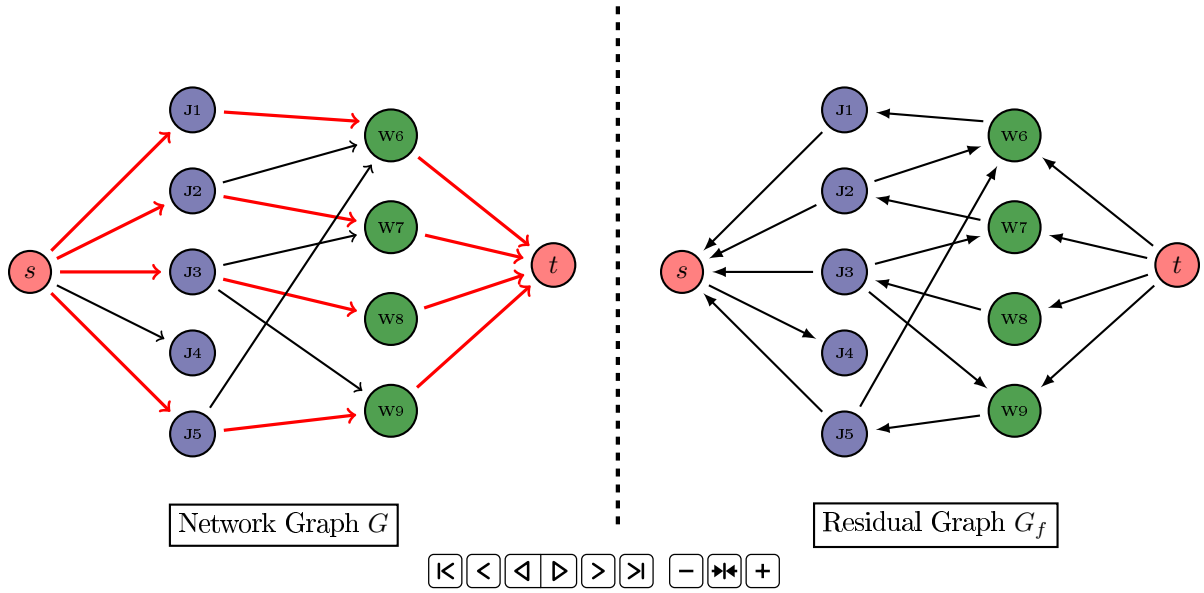


Figure 12.8: Matching Network G

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

12.3.5 An example of weighted graph

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

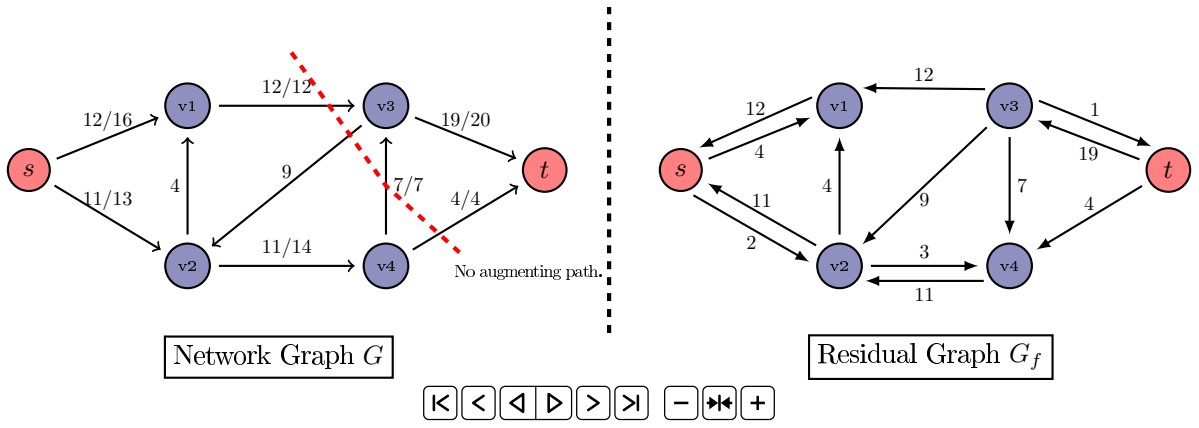


Figure 12.9: A Network Flow Problem. The value of the maximum flow is 23

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**.

12.3.6 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.

12.3.7 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.