Homework 5 - Written and challenge

Pratyay Dutta (Codeforces ID : pdutt005)

March 13, 2024

Contents

1 Union Find

In Union by height, the tree with smaller height is attached to the tree with a bigger height. We essentially see which tree is deeper. Let us consider the following two trees for merging:

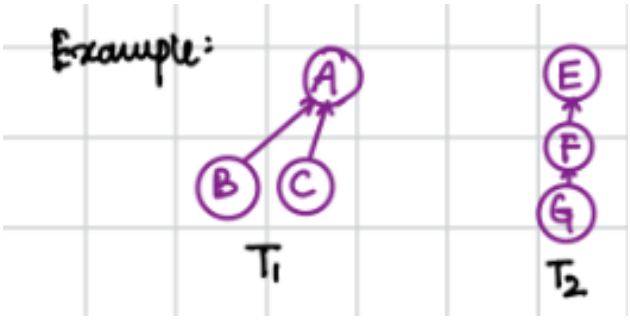


Figure 1: We want to find Union(B,G) through union by height

In this union by height:

- Shallow tree attached to deeper tree.
- Heights of T1 and T2 are h_1 and h_2 and $h_1 < h_2$. Therefore T1 is attached to T2. Height of the unionised tree = h_2

There can be three cases during a union by height:

- $h_1 < h_2$: Height of unionised tree(h_u) = h_2
- $h_1 > h_2$: T2 added to T1 and $h_u = h_1$
- $h_1 = h_2$: Any tree atached to another and $h_u = h_1 + 1$

We see that for h_u to increase by 1, $h_1 = h_2$.
This means that for a weight balanced tree, for the height to increase by 1, the number of elements at least doubles. Therefore, we can come to two conclusions:

1. $h_1 < h_2$ (or $h_1 > h_2$) : $h_u = log(n_2)$:T1 does not increase the height. Therefore:

$$h_u = max(log(n_1), log(n_2)) \tag{1}$$
$$h_u < log(n_1 + n_2) \tag{2}$$
$$h_u = o(log(n)) \tag{3}$$

2. $h_1 = h_2$: $h_u = 1 + log(n_1)$

$$h_u = log(n_1 + n_2) \tag{4}$$
$$h_u = \theta(log(n)) \tag{5}$$
$$\tag{6}$$

From 3 and 5, we can see that $h_u = o(log(n))$ or $h_u = \theta(log(n))$. Therefore:

$$h_u = O(log(n)) \tag{7}$$

2 Network Flow

2.1 Augmenting Path

For the given graph, we first initialise our max flow finding algorithm by considering all flow to be 0. The corresponding residual graph is the graph with the capacities i.e the given graph. We have to find an augmenting path in that graph. The initialised flow graph and the corresponding residual graph is shown in the following image :

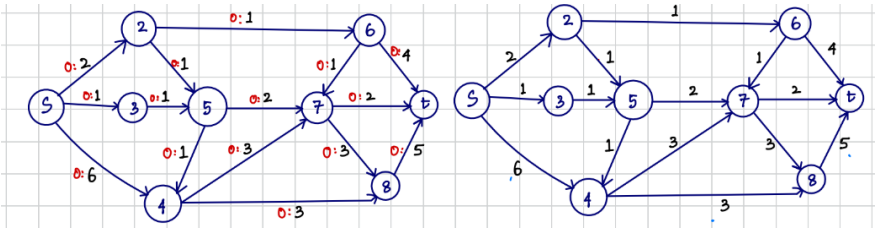


Figure 2: The one on the left is the initialised flow graph. The red represents the flow and the black represents the capacities. the graph on the right is the corresponding residual graph. This representation will be followed throughout the solution.

The augmenting path in the residual graph is : **s 4 8 t**.
The corresponding residual capacity is : min(6,3,5) = 3
Therefore, the flow in this path will be increased by 3. .

2.2 Residual graph

Processing the augmenting path:

- Getting the augmenting path.
- Getting the residual capacity.
- All front edges += 3
- All back edges -= 3

There are no back edges in our residual graph. Therefore, the new flow graph and the residual graph after processing the augmenting path :

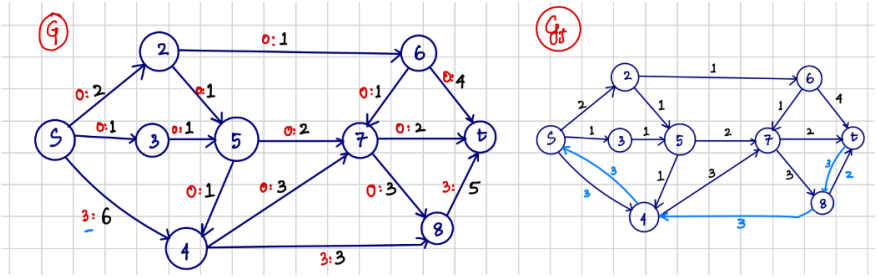


Figure 3: Left : New flow graph. Right : Residual graph

2.3 Ford Fulkerson for max flow

The Ford Fulkerson method for finding the max flow works in the following way:

- $f[u, v] \leftarrow 0$
- Find G_f : The residual graph
- while an augmenting path exists in G_f:
  - Find augmenting path
  - Find residual capacity = $\min(w(u, v))$ in the given path in residual graph (w(u,v) is the weight of an edge)
  - Augment the path.

Once we make a residual graph where no augmenting path exists, it means we have achieved max flow. According to this algorithm, we have already initialised the graph and found an augmented graph in the previous questions as illustrated in 3 and 2, we will continue from there.

1. Run 1 :

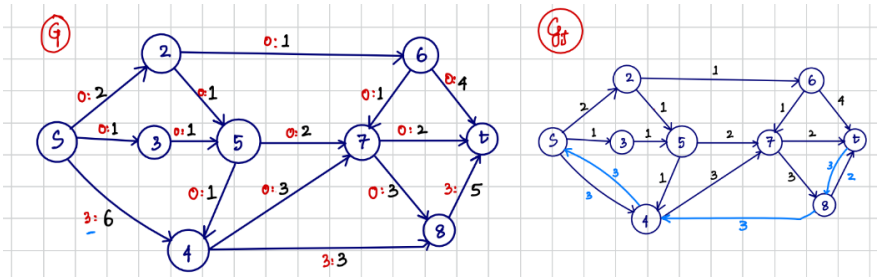


Figure 4: Left : New flow graph. Right : Residual graph

Augmenting path = **s 4 7 t**
Residual Capacity = min(3,3,2) = 2

2. Run 2:

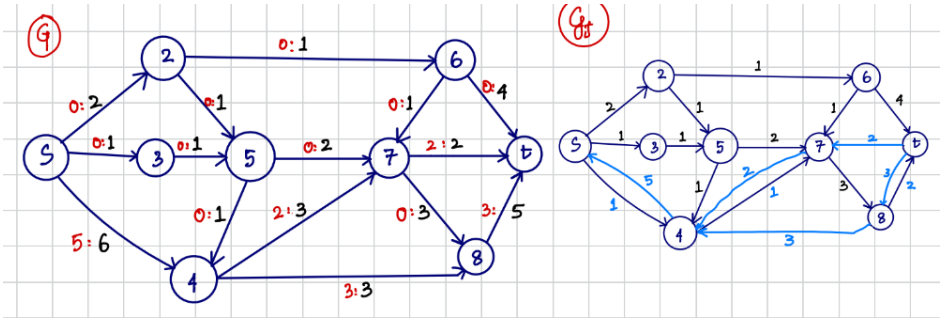


Figure 5: Left : New flow graph. Right : Residual graph

Augmenting path = **s 2 6 t**
Residual Capacity = min(2,1,4) = 1

3. Run 3 :

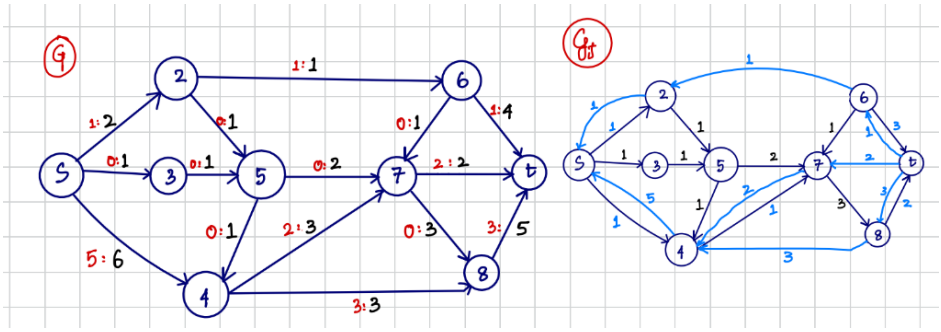


Figure 6: Left : New flow graph. Right : Residual graph

Augmenting path = **s 3 5 7 8 t**
Residual Capacity = min(1,1,2,3,2) = 1

4. Run 4:

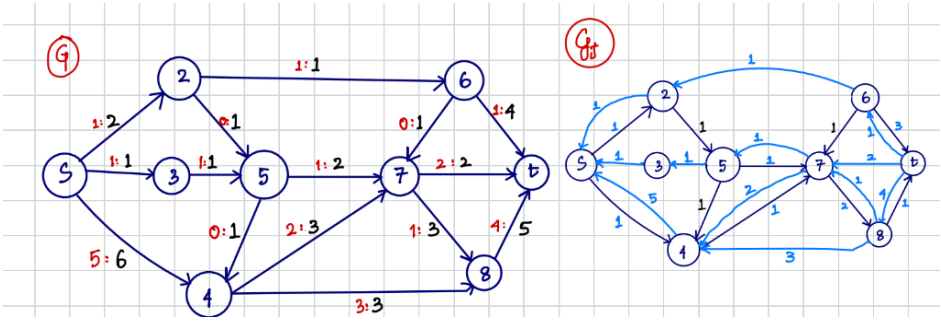


Figure 7: Left : New flow graph. Right : Residual graph

Augmenting path = **s 2 5 7 8 t**
Residual Capacity = min(1,1,1,2,1) = 1

5. Run 5:

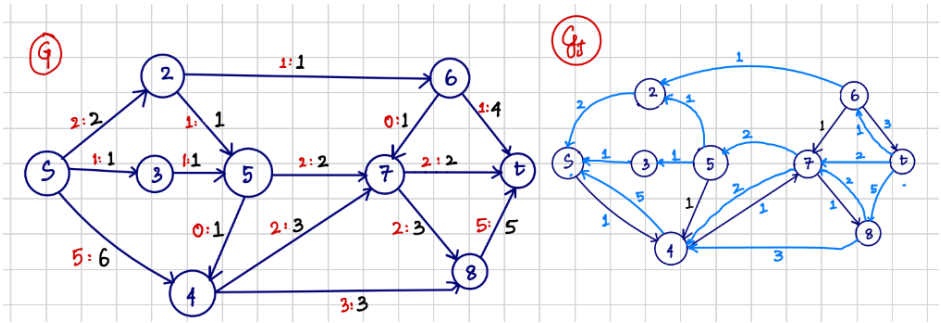


Figure 8: Left : New flow graph. Right : Residual graph

We cannot find any augmenting path in the resulting residual graph.
Therefore, we have achieved max flow.

The max flow in the graph is the total flow coming out of the source (going into the sink).
The max flow for this graph is 8.

3 Bipartite Graph Matching

3.1 Augmenting Path

The augmenting path is : **a 1 b 2 c 5**

3.2 Match

The match : **a-1, b-2, c-5, d-4**. The following is the new matching.

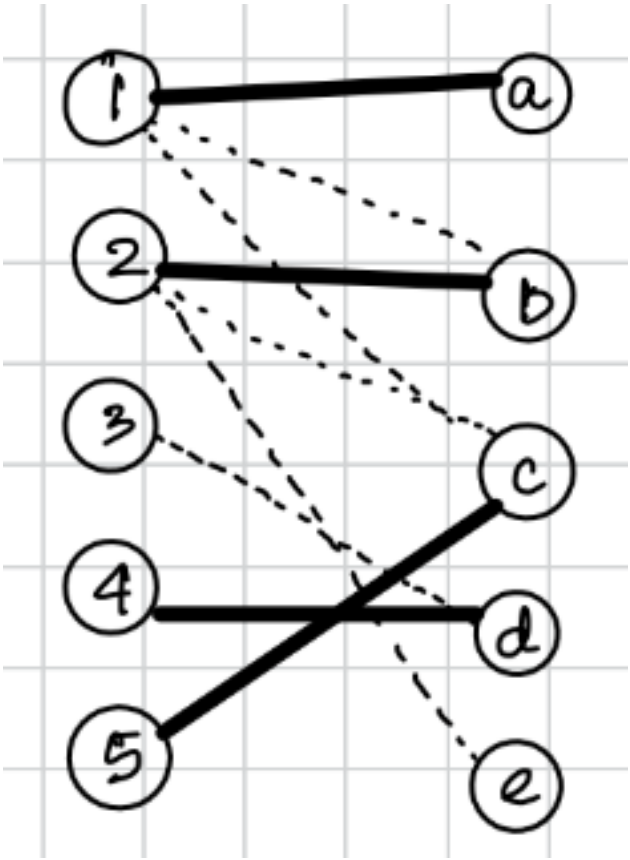


Figure 9: Matching is shown in bold

4 Strongly connected components

4.1 Definition

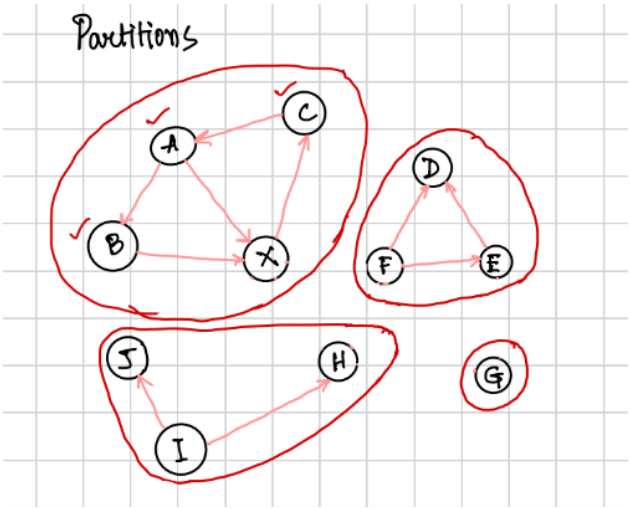
A strongly connected component refers to a segment within a directed graph where there exists a route from each vertex to every other vertex within the same segment.

4.2 Which vertices in same SCC as X

The vertices in the same SCC of vertex X : **X,A,B,C**

4.3 BGSS partition

After running reachability on X:
S_1 = nodes which X can reach = A,B,C,F,E,D
S_2 = nodes which can reach X = A,B,C,J,I,H
SCC (group 1)= $S_1 \cup S_2$ = A,B,C
Reachable from X only (group 2)= $S_1 - S_1 \cup S_2$ = F,E,D
X reachable from (group 3) = $S_2 - S_1 \cup S_2$ = J,I,H
Not connected to X (group 4) = G The following diagram shows the BGSS divisions:



4.4 Number of SCCs

There are 8 SCCs in the given graph. The SCCs are **[E],[F],[G],[H],[A,B,C,X],[I],[J],[D]**

5 Challenge Problems

5.1 Catch the theft - Submission ID : 250842916

The problem asks us to find the minimum number of police vehicles needed to cut off the path for the thief from the beginning to escape. The problem translates to finding the min cut of a graph. According to graph theory, finding the min cut of a graph is equal to finding the max flow of the graph.
In this problem therefore, we have to find the max flow of the graph. To construct our graph, the crossroads are our edges and the number fo police vehicles needed to block that checkpoint is the weight of that edge. We employ the Edmonds Karp variation of the Ford Fulkerson algorithm to find the max flow of the given graph.
Our code performs the following operations :

- Defines a function `bfs` that implements Breadth-First Search (BFS) on a residual graph. It checks if there is a path from a source node `s` to a sink node `t`, and records this path using a `parent` array.
  - Initializes a `visited` list to keep track of visited nodes, marking all nodes as not visited initially.
  - Uses a queue (implemented with `deque`) to explore the graph level by level starting from the source node.
  - Updates the `visited` status of nodes as they are discovered, and records the parent of each node to reconstruct the path later.
  - Returns `True` if the sink node is reachable from the source, indicating that a path exists.
- Defines the `edmonds_karp` function that calculates the maximum flow from the source `s` to the sink `t` in the given graph.
  - Initializes a variable `max_flow` to zero, representing the total flow in the network initially.
  - Creates a residual graph (`rGraph`) as a copy of the original graph to track remaining capacities.
  - Repeatedly finds augmenting paths using `bfs` and updates the `max_flow` with the flow of these paths.
  - For each augmenting path found, it calculates the path's flow as the minimum residual capacity along the path.
  - Updates the residual capacities in the `rGraph` for both forward and reverse edges along the path.
  - Continues this process until no augmenting path can be found from source to sink.
- Ultimately, returns the value of `max_flow`, which is the maximum flow possible from the source to the sink in the network.

Time Complexity : The time complexity of the Edmonds-Karp algorithm can be analyzed as follows:

- The algorithm uses a breadth-first search (BFS) to find the shortest path from the source to the sink in the residual graph. Each BFS operation has a time complexity of $O(E)$, where E is the number of edges in the graph.
- In the worst case, an edge can be part of the augmenting path $O(V)$ times, where V is the number of vertices in the graph. This is because the shortest path found by BFS ensures that the length of the augmenting paths increases monotonically.
- Therefore, the total number of BFS operations performed by the algorithm is $O(VE)$, as each edge can be processed $O(V)$ times across all BFS searches.
- The overall time complexity of the Edmonds-Karp algorithm is $O(VE^2)$.

5.2 Reimbursement - Submission ID : 250878367

In this problem, we have to find the minimum of the maximum cost Yihan can spend on living in hotels during her road trip from Pittsburgh to Riverside keeping in mind her gas constraint. If we consider the cities as nodes and the highways connecting them as edges, this problem can be treated as a variant of the Dijkstra algorithm, where instead of finding the shortest path from the source to the target, we find the least cost path. In Dijkstra, we greedily relax the edge having the least edge weight. In our algorithm, we relax those nodes which minimizes the maximum hotel price while checking if that edge falls in the constraint of the gas cost. The relaxation condition checks both for a lower maximum hotel price and for lower gas usage for the same maximum hotel price, rather than just checking for a lower cumulative distance/cost.
This is how our algorithm works:

1. Initialization:
    - The hotel prices array is adjusted to align with the city indexing by adding a 0 at the beginning, making the indexing start from 1.
    - A priority queue (min-heap) is initialized with a tuple containing the hotel price at the starting city u, the total gas used set to 0, and the current node set to u.
    - A visited dictionary is initialized to track the best (minimum maximum hotel price, minimum total gas used) condition for each node, with all values initially set to infinity. The starting node's condition is updated to reflect its hotel price and 0 gas usage.
2. **Modified Dijkstra's Logic:**
    - The algorithm pops the top element from the priority queue, which represents the path with the current minimum maximum hotel price to reach that node.
    - For each adjacent node (reachable city) of the current node, it checks if the total gas used plus the gas needed to reach the next node does not exceed the gas limit s.
    - It calculates a new maximum hotel price for the path ending at this next node, which is the maximum of the current path's maximum hotel price and the hotel price at the next node.
    - If this new path offers a lower maximum hotel price or a lower gas usage for the same maximum hotel price (when compared to the previously recorded conditions for reaching this next node), the visited dictionary and the priority queue are updated with this better condition.
3. **Termination and Result:**
    - The algorithm terminates once it reaches the destination city v and returns the minimum maximum hotel price found for all paths that satisfy the gas constraint.
    - If the destination city is not reachable within the gas limit, the function returns -1.

Time Complexity : The algorithm's time complexity can be dissected based on its primary operations: initialization, priority queue operations, and the relaxation process within the main loop. Here's a detailed analysis:

1. Initialization:
    - Adjusting the hotel prices array and initializing the priority queue and visited dictionary have a time complexity of $O(n)$, where n is the number of cities.
2. Priority Queue Operations:
    - The algorithm uses a priority queue (min-heap) for managing nodes based on the current minimum maximum hotel price. Each insertion into or extraction from the priority queue has a time complexity of $O(\log n)$, assuming there are at most n elements in the queue at any time.

- The total number of priority queue operations depends on the number of edges in the graph since each edge can potentially lead to a relaxation operation that requires updating the priority queue. If the graph has m edges, the total time complexity due to priority queue operations is $O(m \log n)$.

3. Relaxation Process:

   - During each iteration of the main loop, the algorithm examines outgoing edges from the current node and performs a relaxation step if certain conditions are met. This process has a time complexity proportional to the total number of edges, $O(m)$, multiplied by the cost of updating the priority queue, resulting in $O(m \log n)$ for the entire relaxation process across all nodes.

4. Overall Time Complexity:

   - Combining the initialization and the main loop operations, the overall time complexity of the algorithm is $O(n + m \log n)$. The n term from initialization is dominated by the $m \log n$ term from the main loop, leading to a final time complexity of $O(m \log n)$.

Conclusion: The modified Dijkstra algorithm's time complexity is $O(m \log n)$, where m is the number of edges, and n is the number of cities (nodes). This complexity is derived from the use of a priority queue to manage the exploration of paths within the graph under the given constraints.

5.3 Go Cycling - Submission ID : 250178833

In this problem, we have to find the minimum number of entrances used to reach all exits. If we consider the entrances and exits to be two groups of a bipartite graph, this problem essentially translates to a maximum bipartite matching problem with the minimum number of entrances. It is a maximum matching minimum cover algorithm.
We first create the matchings by determining which exits are reachable from each entrance and subsequently connecting those entrances and exits. We now have to find the maximum matching of this bipartite graph. Our algorithm is as follows:

1. `dfs_iterative(grid, start_x, start_y):`
    - Initializes a grid representation and a starting point (coordinates).
    - Utilizes a stack to implement Depth-First Search (DFS) iteratively, allowing exploration of the grid from the specified starting point.
    - Keeps track of visited cells to prevent revisiting the same cell and accumulates exits reached from the current entrance.
    - For each current cell, checks all four directional neighbors (Right, Down, Left, Up) for potential moves.
    - Movement conditions include being within grid bounds, not having previously visited the target cell, and the target cell's height being lower than the current cell's height.
    - Records exits (cells in the bottom row) reached during traversal.
2. `minimum_entrances(n, m, heights):`
    - Employs the `dfs_iterative` function to map each entrance to reachable exits and vice versa, aiming to cover all exits with the minimum number of entrances.
    - Implements a greedy algorithm to select entrances based on their capability to cover the maximum number of yet-uncovered exits.
    - Updates the sets of used entrances and covered exits continuously until all exits are covered or no additional exits can be covered by the remaining entrances.
    - Returns a tuple indicating whether all exits can be covered (`1` for yes, `0` for no) and the count of either the minimum number of entrances required to cover all exits or the total number of covered exits if not all can be covered.

Time Complexity : The algorithm comprises two major components: the depth-first search (DFS) iterative function and the minimum entrances selection process. Here is the time complexity analysis for each part:

1. DFS Iterative Function:
    - The DFS is performed from each entrance on the top row of the grid, of which there are m entrances.
    - In the worst case, DFS explores every cell in the grid once. The grid has $n \times m$ cells.
    - Therefore, the total time complexity for performing DFS from all entrances is $O(m \times n \times m) = O(nm^2)$.
2. Greedy Selection for Minimum Entrances:
    - After DFS, each entrance is associated with its reachable exits, and vice versa.
    - The greedy selection iterates over all entrances, comparing sets of exits to find the entrance covering the most uncovered exits. This comparison is done in $O(m)$ time for each entrance.

- In the worst case, this process is repeated for every entrance until all exits are covered. Therefore, the worst-case time complexity is $O(m^2)$, considering set operations for each entrance.
- However, since this operation also involves set difference and update operations that could take up to $O(m)$ time each, the worst-case complexity for this part can be considered as $O(m^3)$, assuming each operation on the set takes $O(m)$ and is done for each of m entrances.

3. **Total Time Complexity:**

- Combining both parts, the total time complexity is $O(nm^2 + m^3)$.
- Given that n and m could be of the same order, the complexity simplifies to $O(m^3)$ assuming $m \geq n$ for the sake of simplification.

Conclusion: The time complexity of the algorithm is primarily governed by the DFS from each entrance and the greedy selection process, leading to a worst-case scenario of $O(m^3)$ when considering operations on sets and iterating through all entrances and exits.