Algorithm Assignement 3

Harry CHICHEPORTICHE , Theo DE MORAIS

April 2, 2025

1 Exercise 1 — Basic Graph

1.a — Adjacency Matrix to List and Graph

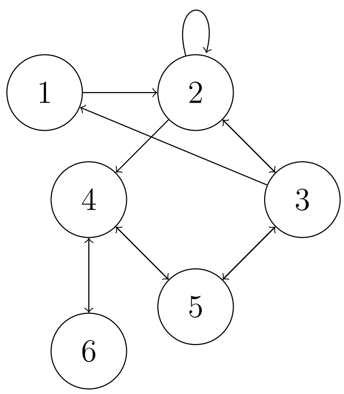
Given adjacency matrix:

$$\text{Adj} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Adjacency list (nodes 1 to 6, in numerical/alphabetical order):

- 1: 2
- 2: 2, 3, 4
- 3: 1, 2, 5
- 4: 5, 6
- 5: 3, 4
- 6: 4

Graphical representation:



Adjacency list from Figure 1 (nodes A to J, alphabetical order):

- A: B
- B: C, D
- C: E, F
- D: E, F
- E: G, F, J
- F: B, G, H, J
- G:
- H: I
- I:
- J: I

1.b — DFS and BFS from vertex A

DFS Traversal (alphabetical order):

- Start at A. Visit: A
- A $\rightarrow$ B. Visit: B
- B $\rightarrow$ C. Visit: C
- C $\rightarrow$ E. Visit: E
- E $\rightarrow$ F. Visit: F
- F $\rightarrow$ G. Visit: G
- F $\rightarrow$ H. Visit: H
- H $\rightarrow$ I. Visit: I
- F $\rightarrow$ J. Visit: J
- B $\rightarrow$ D. Visit: D

DFS Order: A, B, C, E, F, G, H, I, J, D

DFS Timestamps (start/finish):

Node	Start	Finish
A	1	20
B	2	19
C	3	16
E	4	15
F	5	14
G	6	7
H	8	11
I	9	10
J	12	13
D	17	18

Breadth-First Search (BFS):
Starting from vertex A, visiting neighbors in alphabetical order.

- Start at A → Queue: [B]
- Visit B → Enqueue: C, D
- Visit C → Enqueue: E, F
- Visit D → (E, F already visited)
- Visit E → Enqueue: G, J
- Visit F → Enqueue: H
- Visit G → —
- Visit J → Enqueue: I
- Visit H → —
- Visit I → —

BFS Order: A, B, C, D, E, F, G, J, H, I

1.c — Remove one edge to make the graph a DAG and perform topological sort

To transform the graph into a DAG, we must remove an edge that creates a cycle.
From our DFS, we observe that the graph contains cycles such as:

$$F \to B \to C \to E \to F$$

This is a cycle: $F \to B \to C \to E \to F$
To break this cycle, we can remove the edge:

$$\mathbf{F \to B}$$

Once this edge is removed, the graph becomes acyclic.
We then perform a **topological sort** using the finish times from the corrected DFS (reversed order of finish):

- A (20)
- B (19)
- D (18)
- C (16)
- E (15)
- F (14)
- J (13)
- H (11)
- I (10)
- G (7)

Topological order (sorted by decreasing finish time):

A, B, D, C, E, F, J, H, I, G

(Note: B is still present since we removed the back edge to prevent re-visiting it)

1.d — Transform any directed graph into a DAG

We propose the following generic algorithm to remove cycles and convert any directed graph $G = (V, E)$ into a DAG:

Cycle Removal Algorithm

1. Run DFS on the graph and track the recursion stack.
2. During traversal, if a node is revisited while it is still in the recursion stack $\rightarrow$ cycle detected.
3. Identify and remove one of the back edges responsible for the cycle.
4. Repeat this process until all back edges are removed and no cycles remain.

Application to Figure 1 with two new edges:

- $I \rightarrow C$
- $C \rightarrow A$

These additions create the following cycles:

1. $F \rightarrow B \rightarrow C \rightarrow E \rightarrow F$
2. $C \rightarrow A \rightarrow B \rightarrow C$
3. $J \rightarrow I \rightarrow C \rightarrow A \rightarrow B \rightarrow C$

To resolve all cycles: remove the following edges:

- $F \to B$
- $C \to A$
- $I \to C$

Once these are removed, the graph becomes acyclic and topological sorting is possible.

2 Exercise 2 — Cable Network

2.a — Can we connect all nodes within budget $b = 30$?

We aim to connect all the nodes with minimum total cost. This is a classic **Minimum Spanning Tree (MST)** problem.

We use **Kruskal's algorithm**, which adds edges in increasing order of weight while avoiding cycles.

Edges sorted by weight:

- (A, D) = 1
- (C, D) = 2
- (D, E) = 2
- (D, F) = 4
- (B, D) = 4
- (A, B) = 5
- (C, G) = 6
- (F, H) = 7
- (B, H) = 8
- (E, H) = 8
- (F, G) = 9

MST construction:

- Add (A, D) = 1
- Add (C, D) = 2
- Add (D, E) = 2
- Add (D, F) = 4
- Add (B, D) = 4
- Add (C, G) = 6
- Add (F, H) = 7

Total cost: $1 + 2 + 2 + 4 + 4 + 6 + 7 = 26 \leq 30$
Conclusion: Yes, it is possible to connect all neighborhoods within the budget.

2.b — D is restricted to maximum 3 edges

Let's count the number of edges connected to node D in the previous MST:

- (A, D)
- (C, D)
- (D, E)
- (D, F)
- (B, D)

$\rightarrow$ That's 5 edges! Not allowed.
We now rerun Kruskal's algorithm **while ensuring D gets at most 3 edges**.
New MST with constraint on D:

- (A, D) = 1
- (C, D) = 2
- (D, E) = 2 $\rightarrow$ D has now 3 edges
- Avoid (B, D) and (D, F)
- Add (A, B) = 5
- Add (C, G) = 6
- Add (F, H) = 7
- Add (E, H) = 8

Total cost: $1 + 2 + 2 + 5 + 6 + 7 + 8 = 31 > 30$
Conclusion: No, we cannot connect all nodes under the budget if D is restricted to 3 edges. **Also:** this solution is not globally optimal — local constraints on degree break Kruskal's global optimality.

2.c — Edge replacement to meet stricter budget $b' = 25$

We now try to adjust the graph by swapping a single edge.
Suggested swap: replace (F, H) = 7 with (B, H) = 8 $\rightarrow$ That's worse. Let's try:
Replace (C, G) = 6 with a cheaper connection — but no cheaper edge connects G.
Let's test a **cheaper MST overall**:
Alternative MST:

- (A, D) = 1
- (C, D) = 2
- (D, E) = 2
- (D, F) = 4

- (B, D) = 4
- (E, H) = 8
- (C, G) = 6

Total: $1 + 2 + 2 + 4 + 4 + 8 + 6 = 27 > 25$
Try removing (C, G) = 6 $\rightarrow$ now G unconnected.
No matter how we tweak the edges, **we can't get an MST under $b' = 25$**.
Conclusion: No, it is not possible to connect all nodes with a single edge swap if $b' = 25$.

3 Exercise 3 — Finding Champion

3.a — Algorithm to Identify Champions in a Directed Graph

A node u is called a **champion** if it can reach every other node in the directed graph $G = (V, E)$ through a path (direct or indirect).

Goal: Propose an algorithm that identifies and lists all such champions.

Observation: A node u is a champion if and only if a DFS (or BFS) starting from u visits all the nodes in V.

Algorithm:

Champion-Finder Algorithm

1. Let *champions* $\leftarrow \emptyset$
2. For each node u in V:
   (a) Run DFS (or BFS) from u
   (b) If the number of visited nodes equals $|V|$, then u is a champion
   (c) Add u to *champions*
3. Return the set *champions*

Time Complexity: $\mathcal{O}(V \cdot (V + E))$ (We run DFS from each node, and each DFS takes $\mathcal{O}(V + E)$)

Application to the given graph:

- Nodes = {A, B, C, D, E, F, G}
- Edges =
  - A $\rightarrow$ B, D
  - B $\rightarrow$ A, C
  - D $\rightarrow$ B, C
  - C $\rightarrow$ E, F

  - F → E
  - E → G
  - G → F

- Run DFS from each node:
  - DFS(A) visits all nodes
  - DFS(B) visits all nodes
  - DFS(D) visits all nodes
  - DFS(C), DFS(E), DFS(F), DFS(G) do not reach A or B

Output:

$$\boxed{\text{Champions} = \{A,\ B,\ D\}}$$

3.b — Grouping Nodes by Mutual Reachability (SCC Detection)

Goal: Partition the nodes of a directed graph $G = (V, E)$ into groups where each node in a group can reach every other node in the same group (either directly or indirectly).

These groups are called **Strongly Connected Components (SCCs)**.

Solution: Kosaraju's Algorithm

Kosaraju's Algorithm for SCCs

1. Run a DFS on G and record the finishing times of each node.
2. Compute the transpose of the graph G^T (reverse all edges).
3. Run DFS on G^T, in the order of decreasing finishing times from step 1.
4. Each DFS tree in step 3 forms one strongly connected component.

Time complexity: $\mathcal{O}(V + E)$ (Only two passes of DFS and one reversal of edges)

Application to the given graph: Edges:

$$\begin{aligned}
&A \to B,\ D \\
&B \to A,\ C \\
&D \to B,\ C \\
&C \to E,\ F \\
&F \to E \\
&E \to G \\
&G \to F
\end{aligned}$$

SCCs detected:

- {A, B, D}
- {C}

- {E, F, G}

Conclusion: The graph can be partitioned into the following mutually reachable groups:

$$\boxed{\{\{A, B, D\},\ \{C\},\ \{E, F, G\}\}}$$

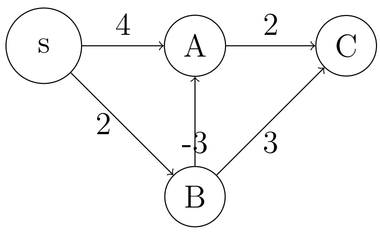
4 Exercise 4 - Shortest Path Problem

Problem Statement

We consider a directed graph $G = (V, E)$ where each edge (u, v) is assigned a weight that can be positive, zero, or negative. The objective is to demonstrate an example where Dijkstra's algorithm fails to correctly compute the shortest path distances from a source vertex s.

Graph Example

Consider the following graph with the given weights:



Expected Result and Dijkstra's Failure

The correct shortest distances from s should be:

- $d(s, A) = -1$ (via $s \to C \to A$)
- $d(s, B) = 2$
- $d(s, C) = 2$
- $d(s, D) = 2$ (via $s \to C \to D$)

However, Dijkstra's algorithm, based on a greedy update strategy, assumes that already computed partial distances are optimal, which is not the case with negative weights. It will incorrectly identify $d(s, A) = 4$, which is incorrect.

Solution: Bellman-Ford Algorithm

To resolve this issue, we can use the Bellman-Ford algorithm, which supports negative weights and detects negative-weight cycles. It runs in $O(VE)$ and ensures correct distance calculations.

Conclusion

Dijkstra's algorithm fails in the presence of negative-weight edges because it does not revisit nodes after their relaxation. Bellman-Ford is a better approach for handling such graphs.

5 Exercise 5 - Maximum Flow

Problem Statement

Given the flow network G shown in Figure 1, we solve the following:

1. Resolve the antiparallel edge issue in G.
2. Walk through the Ford-Fulkerson algorithm by hand.
3. Identify the bottleneck using cuts.
4. Analyze the running time and suggest improvements.

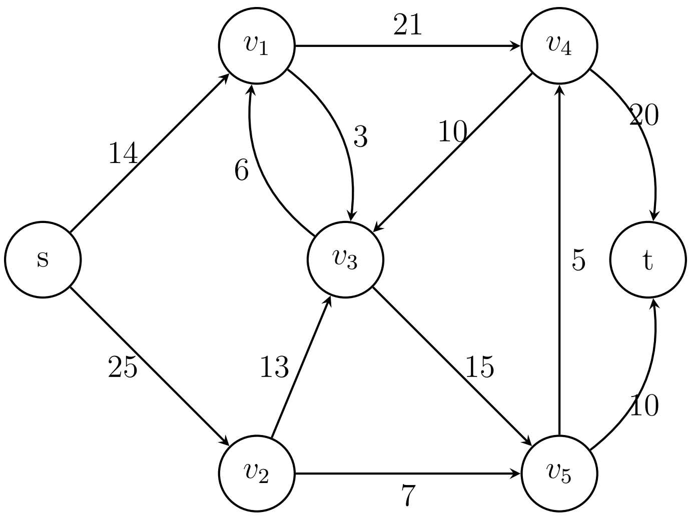


Figure 1: Flow Network G

Resolving the Antiparallel Edge Issue

The antiparallel edges between v_1 and v_3 are resolved by introducing an intermediate node w such that:

- $v_1 \to w$ with capacity 3.
- $w \to v_3$ with capacity 3.
- $v_3 \to w$ with capacity 6.
- $w \to v_1$ with capacity 6.

This removes direct bidirectional edges while maintaining flow constraints.

Applying the Ford-Fulkerson Algorithm

We apply the algorithm to compute the maximum flow:

1. Initial augmenting path: $s \to v_2 \to v_5 \to t$ with flow 7.
2. Next path: $s \to v_1 \to v_4 \to t$ with flow 14.
3. Next path: $s \to v_2 \to v_3 \to v_5 \to t$ with flow 8.
4. Next path: $s \to v_1 \to v_3 \to v_5 \to t$ with flow 3.
5. No more augmenting paths exist.

Total maximum flow: $7 + 14 + 8 + 3 = 32$.

Identifying the Minimum Cut

A valid minimum cut is $S = \{s, v_1, v_2, v_3\}$ and $T = \{v_4, v_5, t\}$, with cut capacity:

$$\begin{aligned} c(S,T) &= c(v_1, v_4) + c(v_2, v_5) + c(v_3, v_5) \\ &= 21 + 7 + 4 = 32. \end{aligned}$$

This confirms the maximum flow found using Ford-Fulkerson.

Running Time and Improvements

Ford-Fulkerson runs in $O(E \cdot f_{max})$. Using BFS (Edmonds-Karp) improves the complexity to $O(VE^2)$, which is better for large graphs.