**John Maurice Hortua**

**Santiago Arboleda Velasco**

**Samuel Viviescas Carrillo**

# **Problem Context**

Students enrolled in the Algorithms and Discrete Structures course at Icesi University are facing challenges in understanding and applying graph theory concepts. The difficulty arises from the abstract nature of these concepts, making their application less intuitive. Additionally, students are confronted with a task where they are required to design and implement a game with a minimum of 50 vertices and 50 edges. The interesting aspect is that students have the opportunity to propose the game themselves.

# **Solution development**

In order to solve this problem, the engineering method was chosen for developing the solution, following a systematic approach that aligns with the stated problem. The following flowchart was defined, and we will follow its steps in the process of the development of the solution.

# **Step 1. Problem identification**

Needs assessments

* The solution must ensure the use of 2 algorithms, whether they are Graph Traversals (BFS, DFS), Minimum Weight Paths (Dijkstra, Floyd-Warshall), or Minimum Spanning Tree (MST) (Prim, Kruskal).
* The solution to the problem must ensure the automatic switch to at least 2 different versions of algorithm implementations.
* The solution to the problem must ensure a user-friendly graphical interface.
* The solution should provide assistance and instructions to the user.

**Problem definition**

Students in the Algorithms and Discrete Structures course at Icesi University must design and implement a game with a minimum of 50 vertices and 50 edges, with the added twist that students can propose the game themselves.

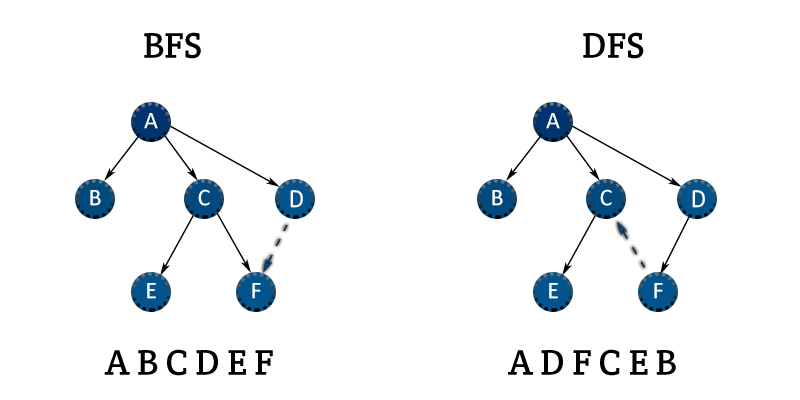
# **Step 2. Background research**

*Definitions*

**BFS**

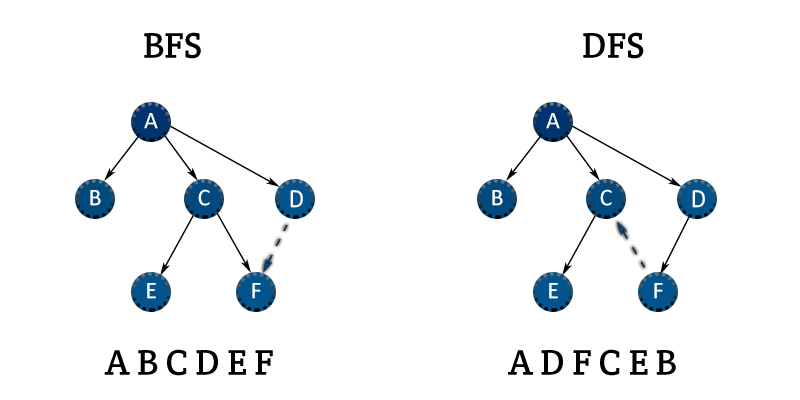
Is a method for graphing data, search trees, and traversing structures. The method visits & marks all critical nodes in a network in an exact breadthwise manner. This algorithm chooses a single node (the beginning or origin point) in a network and visits all nodes near the chosen node.

After visiting and marking the beginning node, the algorithm goes on to the next uninhabited nodes and analyzes them. All nodes are indicated once they have been visited. These cycles continue until all nodes in the graph have been visited and marked correctly. BFS is an abbreviation for Breadth-first search.



**DFS**

Is a depth-first search strategy that may be used to locate or explore graphs or trees. Before backtracking, the algorithm begins at the tree’s root and explores each path. When an iteration reaches a dead end, a stack information structure is utilized to remember, identify the next vertex, and begin a search. DFS’s complete form is a depth-first search.

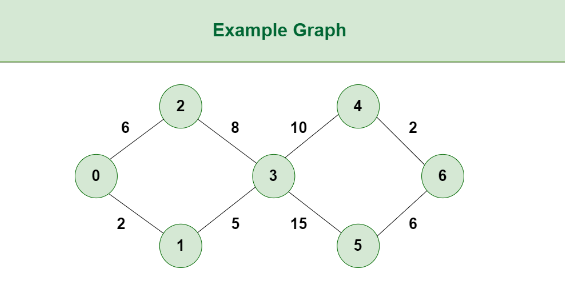


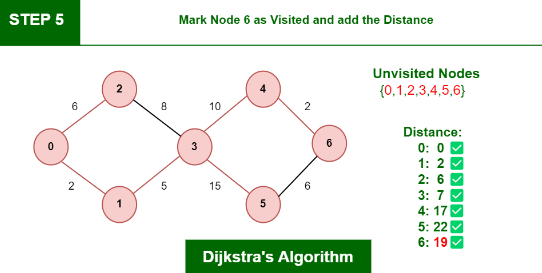
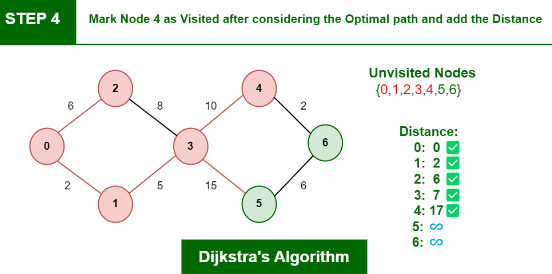
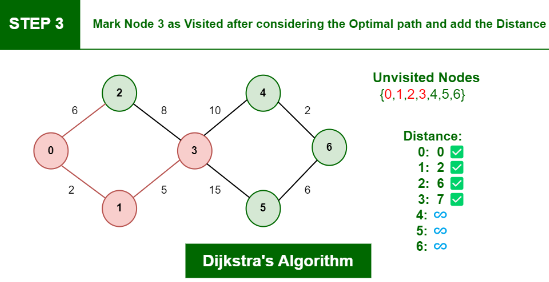
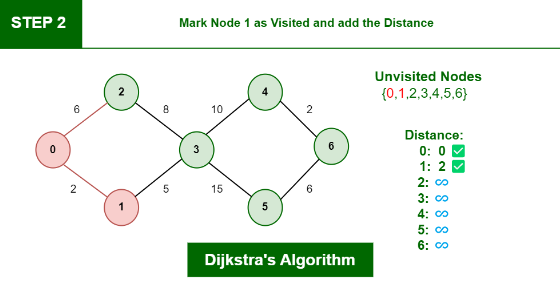
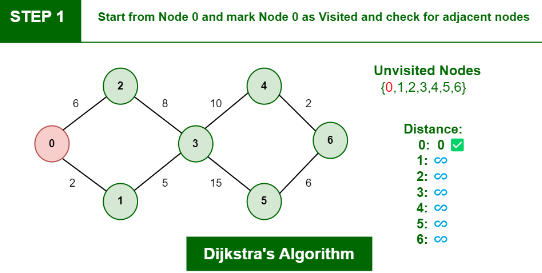
**Dijkstra**

Is a popular algorithm for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist Edsger W. Dijkstra in 1956.

The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source.

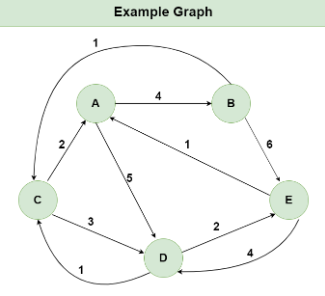
It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

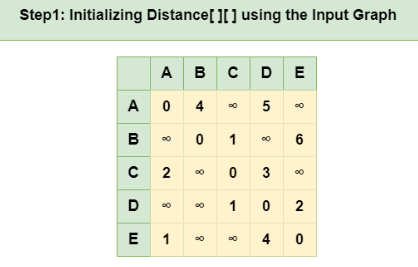


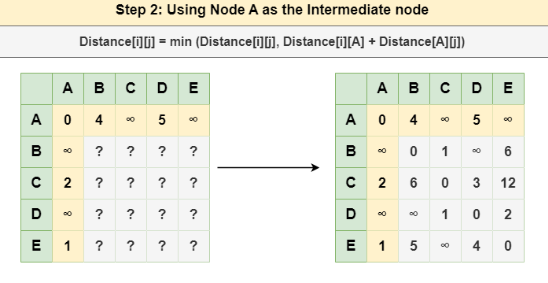


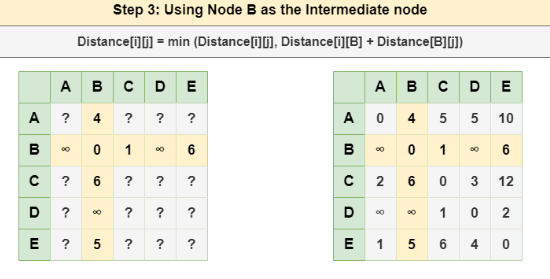
**Floyd-Warshall**

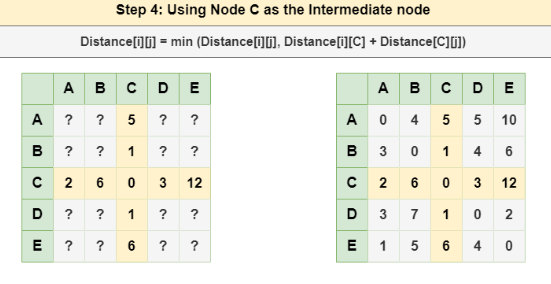
Is a fundamental algorithm in computer science and graph theory. It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both positive and negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems.

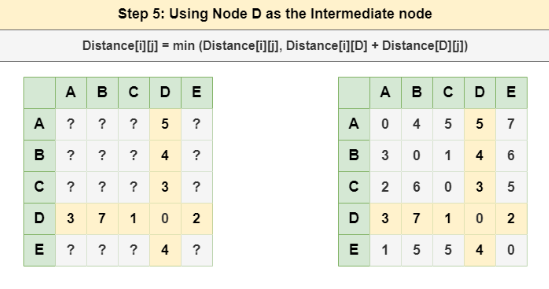
****

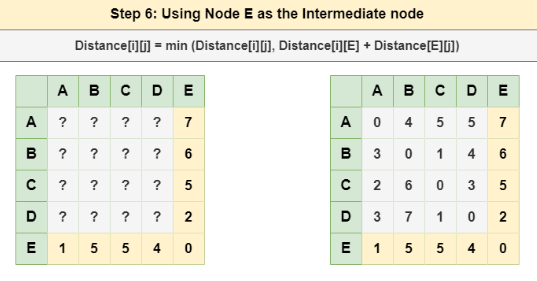
****

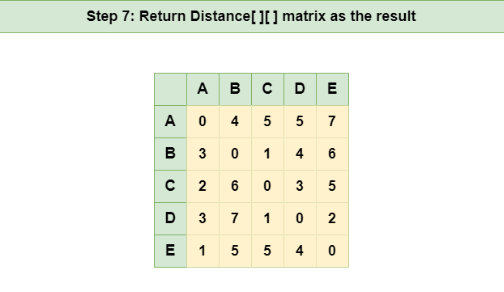
****

****

****

****

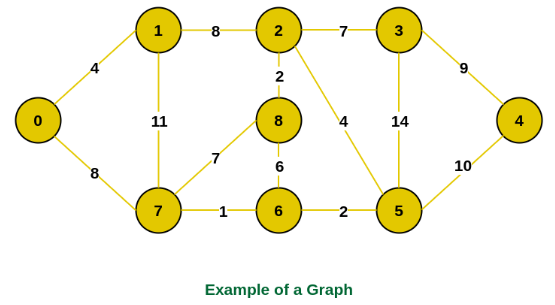
****

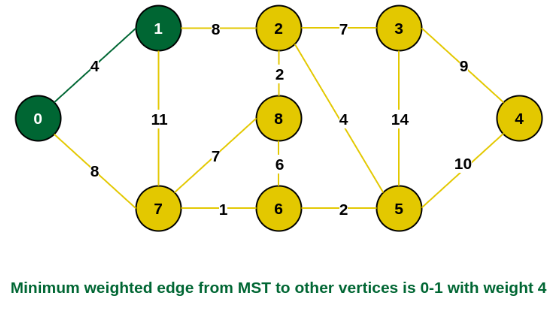
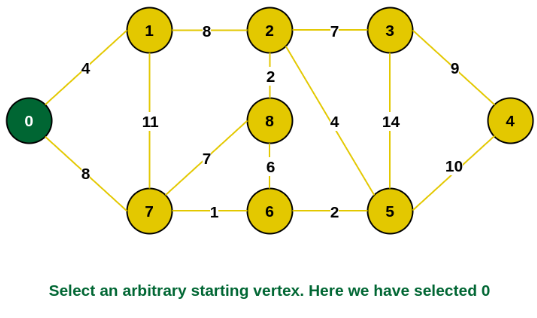
****

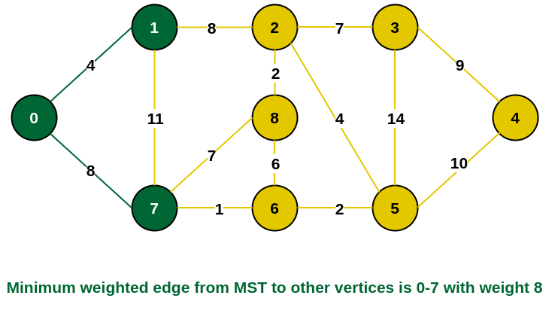
**Prim**

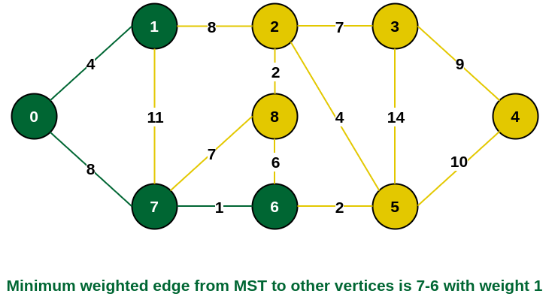
This algorithm always starts with a single node and moves through several adjacent nodes, in order to explore all of the connected edges along the way.

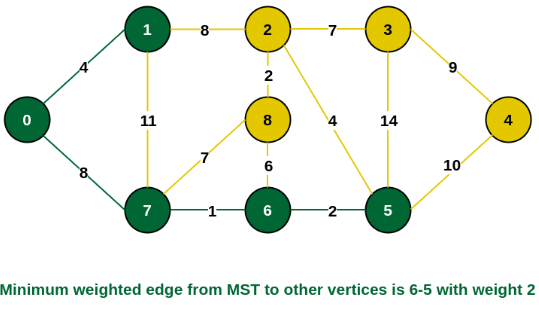
The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

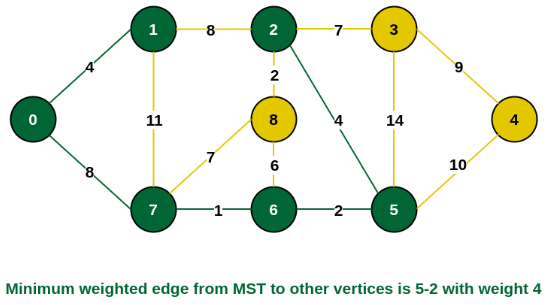
****

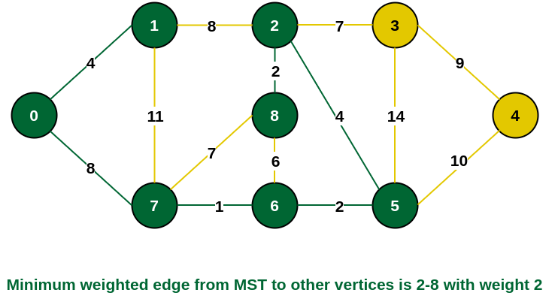
****

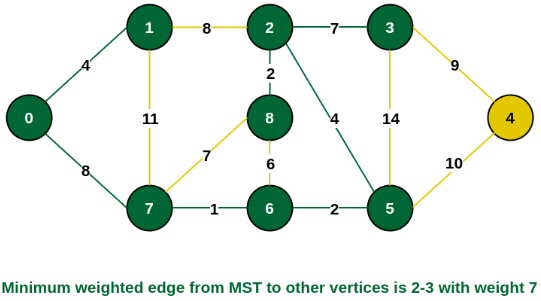
****

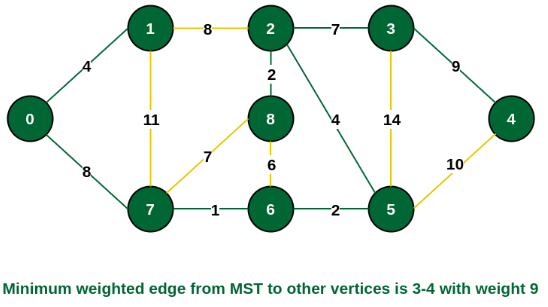
****

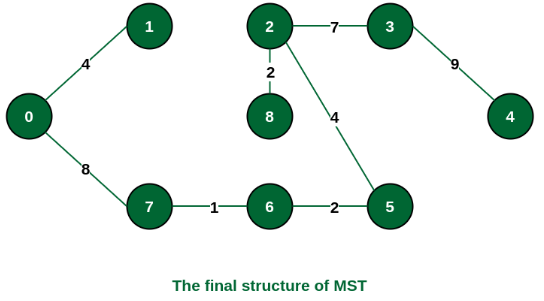
****

****

****

****

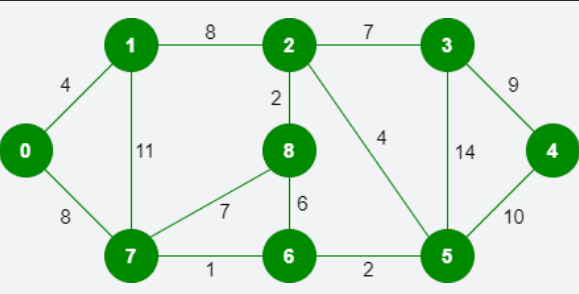
****

****

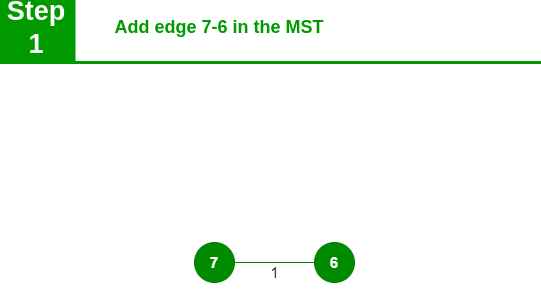
**Kruskal**

In Kruskal’s algorithm, sort all edges of the given graph in increasing order. Then it keeps on adding new edges and nodes in the MST if the newly added edge does not form a cycle. It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution.

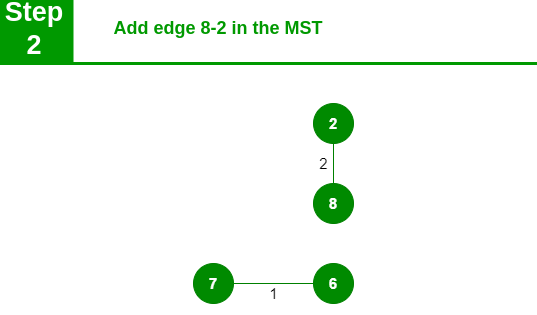
Input Graph:



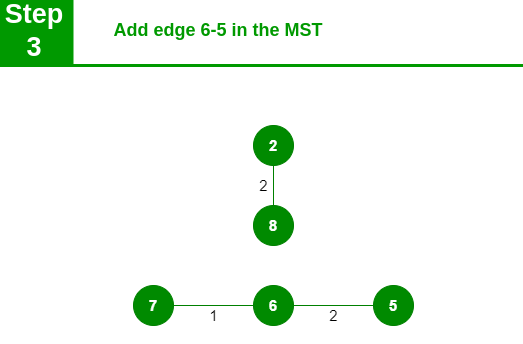
**Step 1: Pick edge 7-6. No cycle is formed, include it.**

****

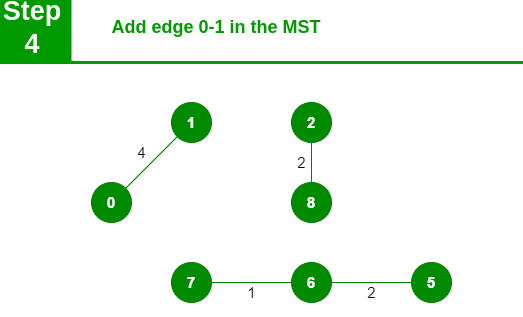
**Step 2: Pick edge 8-2. No cycle is formed, include it.**

****

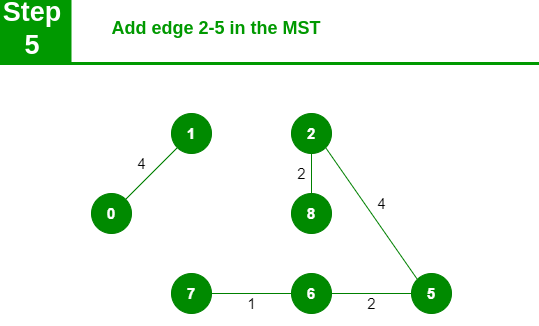
**Step 3: Pick edge 6-5. No cycle is formed, include it.**

****

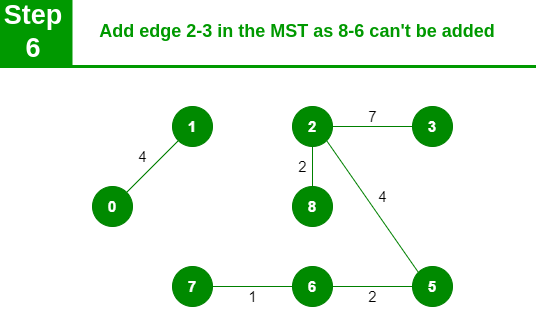
**Step 4: Pick edge 0-1. No cycle is formed, include it.**

****

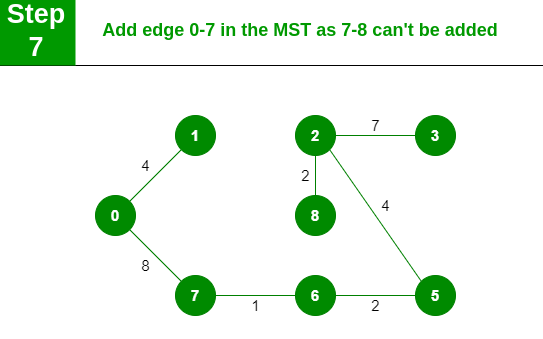
**Step 5: Pick edge 2-5. No cycle is formed, include it.**

****

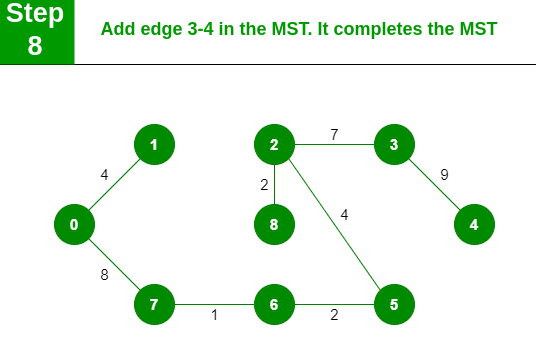
**Step 6: Pick edge 8-6. Since including this edge results in the cycle, discard it. Pick edge 2-3: No cycle is formed, include it.**

****

**Step 7: Pick edge 7-8. Since including this edge results in the cycle, discard it. Pick edge 0-7. No cycle is formed, include it.**

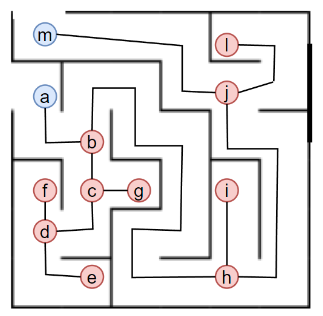
****

**Step 8: Pick edge 1-2. Since including this edge results in the cycle, discard it. Pick edge 3-4. No cycle is formed, include it.**

****

# **Step 3. Search solutions**

**Alternative 1: Maze with Graphs:**

****

**Conceptual Design:**

In this maze game, the labyrinth is represented as a graph where nodes correspond to intersections and edges represent paths. The goal is for players to navigate from the entrance to the exit by traversing the graph. Each intersection (node) presents a decision point, and players must choose the correct path to advance.

**Implementation:**

**Graph Representation:**

Design a graph where each intersection is a node, and each path between intersections is an edge. The graph can be implemented as an adjacency list or matrix.

**Node Content:**

Each node may contain information such as clues, obstacles, or challenges. Players must make decisions at each intersection based on this information.

**Player Movement:**

Players move through the maze by selecting edges that connect nodes. Incorrect choices may lead to dead ends or setbacks, while correct choices bring them closer to the exit.

**Challenges and Puzzles:**

Introduce challenges at certain nodes, such as puzzles that must be solved to proceed. These puzzles can be related to graph theory concepts, adding an educational element.

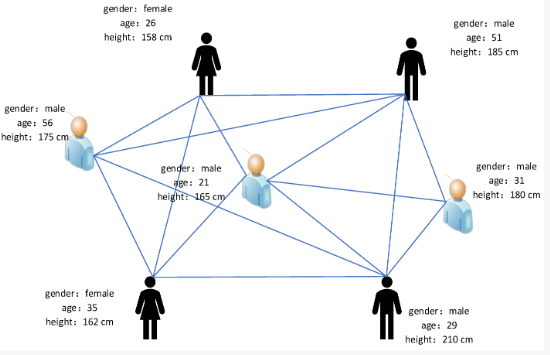
**Game Progression:**

As players progress, the maze complexity can increase. Shortcuts, hidden paths, or teleportation nodes can be introduced to keep the game engaging.

**Goal:**

The ultimate goal is for players to reach the exit node. The game can have multiple levels with increasing difficulty, each represented by a different maze graph

**Alternative 2: Social Network Simulator:**



**Conceptual Design:**

In this simulator, the social network is modeled as a graph, where nodes represent individuals and edges denote relationships. Players engage in activities to influence and shape the dynamics of the network.

**Implementation:**

**Graph Representation:**

Represent individuals as nodes and relationships (friendships, alliances, etc.) as edges. Attributes like popularity, influence, and interests can be associated with each node.

**Player Actions:**

Players can perform actions such as making new connections, influencing others, or participating in events. Each action affects the social graph, altering relationships and attributes.

**Goals and Objectives:**

Assign goals to players, such as becoming the most influential individual, creating the largest social group, or initiating and resolving conflicts within the network.

**Dynamic Events:**

Introduce dynamic events that impact the entire network, such as trending topics, rivalries, or external influences. Players must adapt their strategies to these changes.

**Social Challenges:**

Implement challenges that players can solve using their understanding of the social graph. For example, resolving conflicts, mediating disputes, or fostering positive interactions.

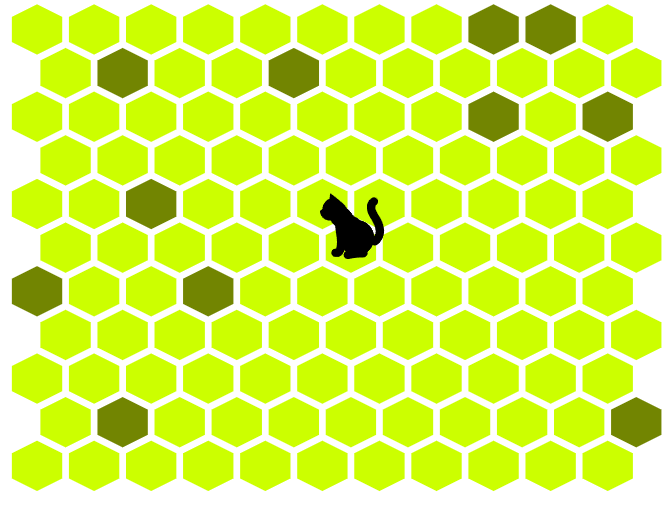
**Feedback System:**

Provide feedback to players on the consequences of their actions. The social graph evolves based on their decisions, affecting the overall dynamics of the simulated network.

**Simulation Depth:**

Increase the complexity by simulating various social aspects like trust, loyalty, and evolving interests. This adds depth to the gameplay and encourages strategic thinking.

**Alternative 3: Trap the cat:**

****

**Conceptual Design:**

In "Trap the Cat," the player aims to strategically encircle a cat that moves across a grid-based board. The challenge lies in selecting adjacent cells to gradually limit the cat's movement until it's trapped.

**Implementation:**

**Grid Representation:**

Design a grid-based board where each cell represents a possible position for the cat. The cat can move freely across navigable cells.

**Cat Movement:**

The cat moves from one cell to an adjacent, navigable cell during its turn. The movement can follow a specific pattern.

**Player Actions:**

The player can select cells to render impassable for the cat. These chosen cells gradually restrict the cat's movement options. Player strategize to predict the cat's next move.

**Trap Formation:**

Player must create a perimeter around the cat. The objective is to limit the cat's accessible cells until it has no valid moves left, leading to its capture.

**Winning Conditions:**

The game can have different winning conditions, such as capturing the cat within a certain number of turns or surviving a set number of rounds without the cat escaping.

# **Step 4. Transition from Ideas to Initial designs**

All the games defined earlier could be modeled using graphs, where each of them has a different objective and functioning. Upon conducting a detailed review of all of the alternatives, we have the following aspects:

**Alternative 1:**

**Alternative 2:**

**Alternative 3:**

# **Step 5. Evaluate and choose solution**

Criteria:

Evaluation:

| Alternatives | Criterion A | Criterion B | Criterion C | Criterion D | Total |
| --- | --- | --- | --- | --- | --- |
| **Alternative** | Approximate  1 | No  1 | More than one  2 | None  1 | 5 |
| **Alternative** | Exact  2 | Yes  2 | All  3 | High  2 | 9 |

**Solution selection:**

# **Step 6. Reports and specifications**

Problem Specification

**ADTs**

| **ADT Stack** |
| --- |
| **Stack** = {item3, item2, item1} |
| **inv:** Last in first out |
| **Basic operations:**   * **push** Stack x Element-> item * **pop** Element -> item * **peek** Element -> item * **isEmpty** Element -> boolean |

**pop()**

“Pops and returns the item on the top of the stack.”

{pre: stack.isEmpty() == false}

{post: stack.size() = stack.size() -1 }

**peek()**

“Returns the item at the top of the stack without removing it.”

{pre: stack.isEmpty() == false}

{post: true}

**isEmpty()**

“Returns true if the stack contains no elements.”

{pre: true}

{post: true }

| **ADT Heap** |
| --- |
|  |
| **inv: (**valueParentNode >= valueChildrenNode) V  **(**valueParentNode <= valueChildreNode) |
| **Basic operations:**   * **peek** Heap -> item * **size** Heap -> int * **add** Heap x item -> item * **poll** Heap -> item * **swap** Heap x (int, int) -> Heap * **sort** Heap -> Heap * **siftDown** Heap x (BiPredicate, int, int) -> Heap * **siftUp** Heap x int -> Heap * **isEmpty** Heap -> boolean |

**peek()**

“ Returns the element at the top of the heap without removing it.”

{pre: heap.isEmpty() == false}

{post: true}

**size()**

“Returns the number of elements currently in the heap.”

{pre: heap.isEmpty() == false}

{post: true}

**add(T item)**

“Adds an element to the heap.”

{pre: true}

{post: heap invariant (max, min heap) is still valid}

**poll()**

“Removes and returns the root element of the heap.”

{pre: heap.isEmpty() == false}

{post: heap.get(0) is removed, heap invariant still valid }

**swap(int i, int j)**

“Swaps two elements in the heap array by index.”

{pre: indices must be valid}

{post: heap invariant still valid, values indices have been exchanged}

**sort()**

“Sorts the elements in the heap using heapsort.”

{pre: valid heap invariant }

{post: elements are rearranged}

**siftDown(BiPredicate<T,T> comparator, int index, int size)**

“Sift down operation to maintain heap order property.”

{pre: valid heap invariant}

{post: element has been moved to its correct position}

**siftUp(int index)**

“Elements are compared with their parent and swapped if necessary”

{pre: element is a part of the heap}

{post: element at given position is less than or equal to its parent}

**isEmpty()**

“Returns whether the heap is empty.”

{pre: true}

{post: true}

| **ADT PriorityQueue** |
| --- |
| **PriorityQueue =** {(Task<title, descrp, …, priority>), (Task<title, descrp, …, priority>)} |
| **inv:** (priority == 0 V priority == 1) |
| **Basic operations:**   * **peek** Heap -> item * **size** Heap -> int * **add** Heap x item -> item * **poll** Heap -> item * **isEmpty** Heap -> boolean |

**peek()**

“ Returns the element at the top of the heap without removing it.”

{pre: heap.isEmpty() == false}

{post: true}

**size()**

“Returns the number of elements currently in the heap.”

{pre: heap.isEmpty() == false}

{post: true}

**add(T item)**

“Inserts an item into the priority queue.”

{pre: true}

{post:new element is added to the PriorityQueue in a position that maintains the priority order. }

**poll()**

“Retrieves and removes the head of this queue, or returns null if this queue is empty.”

{pre: true }

{post: element with the highest priority is removed from the PriorityQueue.}

**isEmpty()**

“Returns whether the heap is empty.”

{pre: true}

{post: true}

| **ADT Hash Table** |
| --- |
| **Set of keys:** K = {k1, k2, k3}  **hash function:** h: k->t, t = table position |
| **inv:** size >= 0, values != null |
| **Basic operations:**   * **add** LinkedList x (key, value) -> entry * **remove** LinkedList x key -> void * **key** LinkedList x value -> entry * **values** LinkedList -> LinkedList * **getIndex** LinkedList x key -> int * **get** LinkedList x (bucket, key) -> entry * **get** V x key -> value * **size** int -> int * **peek** V -> value * **hash** int x key -> int * **getLoadFactor** double -> double |

**add(K key, V item)**

“Adds the new key-value pair to the hash table”

{pre: true}

{post: key-value pair is added to the hash table.}

**remove(K key)**

“removes the entry from the linked list”

{pre: key is valid}

{post: If the key exists it has been removed, along with its associated value.}

**key(V value)**

“search for a key, based on a given value in the hashtable”

{pre: value != null}

{post: if value is in the hashtable return corresponding key, else null}

**values()**

“collects and returns all the values stored in the hash table”

{pre Each linked list within the hash table contains entries of type entry<K, V>:}

{post: returns a linked list that contains all the values from the hash table.}

**getIndex(K key)**

“calculates an index in a hash table based on the hash code of a given key.”

{pre: key != null}

{post: returns an index within the bounds of the hash table (0 <= index < table.length).}

**get(LinkedList<Entry<K,V>> bucket, K key)**

“retrieve the entry associated with a given key”

{pre: key != null}

{post: If the key is found returns the associated value, else return null }

**get(K key)**

“retrieve the value associated to an entry given a key”

{pre: key != null}

{post: If the key is found returns the associated value, else return null }

**hash(K key)**

“hashes the given key”

{pre: key != null}

{post: returns the result of hashing the key }

# **Step 7. Implementation**

Solution implementation in Java:

**List of task to implement:**

* Create Task
* Create Reminder
* Edit task
* Delete task
* Undo

**Subroutine specifications**

| Name: | Create Task |
| --- | --- |
| Description: | allows to create a task |
| Input: | - title: String, title of the task  -description: String, what is the purpose of the task  -deadline:LocalDateTime, date to finish the task  -newHasPriority: Priority, if the task has priority or not |
| Output | Task , task with all parameters setted |

Implementation

| Name: | Create Reminder |
| --- | --- |
| Description: | allows to create a reminder |
| Input: | - title: String, title of the reminder  -description: String, what do you want to remind later  -deadline:LocalDateTime, date to remind  -hasPriority: Priority, if the reminder has priority or not |
| Output | Reminder, reminder with all parameters setted |

implementation

| Name: | Edit |
| --- | --- |
| Description: | allows to edit a task |
| Input: | - title: String, title of the task that is going to edit  -index: int, task index  -newTitle: String, new task title  -newDescription: String, new description  -newDeadline:LocalDateTime, new due date  -newHasPriority: Priority, if the task has priority or no |
| Output | Task , task with new parameters setted |

implementation

| Name: | Delete |
| --- | --- |
| Description: | allows to delete a task |
| Input: | -taskTitle: String, title of the task to delete  -index: int, index of the task |
| Output | nill , task deleted |

| Name: | Undo |
| --- | --- |
| Description: | allows to reverse the most recent action |
| Input: |  |
| Output | nil, restored last command |

**References**

​​*Difference Between BFS And DFS*. (n.d.). Retrieved from<https://unacademy.com/content/gate-cse-it/difference-between-bfs-and-dfs/>

*​Floyd Warshall Algorithm - GeeksforGeeks*. (n.d.). Retrieved from<https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/>

*​Kruskal’s Minimum Spanning Tree (MST) Algorithm - GeeksforGeeks*. (n.d.). Retrieved from<https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>

*​Prim’s Algorithm for Minimum Spanning Tree (MST) - GeeksforGeeks*. (n.d.). Retrieved from<https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/>

*​What is Dijkstra’s Algorithm? | Introduction to Dijkstra’s Shortest Path Algorithm - GeeksforGeeks*. (n.d.). Retrieved from<https://www.geeksforgeeks.org/introduction-to-dijkstras-shortest-path-algorithm/>