PHASE 1: PROBLEM IDENTIFICATION

Context of problem:

The company JAV a.s faces the challenge of developing a video game called Blitz Bomb, which involves creating a graph with at least 50 vertices, where some vertices contain bombs and the edges represent gunpowder paths. The goal of the game is for the player, in the role of Kelvin, to efficiently traverse the graph, activating all the bombs and reaching the exit within a limited time, which varies depending on the selected difficulty level (easy, medium, or hard). The game must implement at least two (2) of the graph algorithms.

Options: Paths over Graphs (BFS, DFS), Minimum Weight Paths (Dijkstra, Floyd-Warshall), Minimum Covering Tree -MST- (Prim, Kruskal).

Problem definition: The need to create a game using the graph data structures where the main goal of the character is to make some bombs explode in a set amount of time.

Classification of need:

- Develop a video game called Blitz Bomb.
- Create a game that involves Kelvin, an alien who must activate interconnected bombs in a gunpowder network.
- Implement a simple graph with at least 50 vertices, where some vertices contain bombs and the edges represent gunpowder paths.
- Ensure the graph is connected and has no isolated vertices.
- Calculate the minimum time for Kelvin to traverse the graph to activate all the bombs and reach the exit.
- Establish three difficulty levels: easy, medium, and hard, with different time limits to complete the game.
- Implement two versions of the game using matrix adjacency and adjacency list data structures, with the ability to be interchangeable without affecting game functionality.
- Use two graph algorithms, for the game's operation.
- The game must contain an attractive graphic interface.

REQUIREMENTS:

FR1. User registration:

The software should allow users to register in the game by providing a nickname

FR2. Menu

The software should display a menu to the user that includes the following options: "Play," "Difficulty," and "Graph Type." When the user selects the "Difficulty" option, three additional choices become available: "Easy," "Medium," and "Hard."

FR3. Init-Gameplay

The software should allow the user to view the game by displaying a randomly generated graph. It will also present the estimated time using an animated panel that functions as a countdown timer, with the feature that the time decreases in real-time.

FR4. Graph generation

The software should allow the user to select the graph implementation they wish to use, whether it be via an adjacency list or an adjacency matrix.

Once the play option has been selected, the software will automatically generate a random simple graph consisting of a minimum of 50 vertices, each vertex of the graph shall have at most 4 edges. Some of these vertices will contain bombs, which will be visually represented by bomb images on the corresponding nodes.

FR5. Time calculation

The software will use graph algorithms to calculate the time in which the user will have to run the graph.

FR6. Avatar movement

The software should allow the user to navigate the avatar from one vertex to another using the following keys: A (left), D (right), S (down), W (up).

FR7. Dijkstra power-up

The software must enable the player to use a power-up based on the Dijkstra algorithm. This power-up can be employed by the player to determine the quickest route between two vertices, to do this the software must allow the user to select 2 vertices of the graph

FR8. Save Score

The system must be able to store player scores in a CSV file using the serialization process. This file will contain two separate columns: one for player names and another for the time it took them to complete the game.

Non-Functional Requirements (NFRs):

NFR1. The user interface must be intuitive and user-friendly, catering to players of experience levels.

NFR2. The game must be developed using JavaFX as the primary technology for building the user interface and graphical components.

NFR3. The game must incorporate two different graph implementations, namely an adjacency matrix data structure and an adjacency list data structure. These two graph implementations should be interchangeable without affecting the game's functionality.

The requirements specification can be found in another file inside the DOCS folder.

PHASE 2: GATHERING THE NECESSARY INFORMATION

To gain a comprehensive understanding of the concepts involved in the development of the task and reminder management system, a search for definitions and key elements related to the stated problem was conducted.

Vertex: In the world of graph theory, a vertex, often called a node, is like a basic building block for creating graphs. A graph can be either undirected or directed and is made up of these vertices along with edges (connections between vertices). Imagine a vertex as a point that you can label, usually represented as a circle with a name. An edge, on the other hand, is like a line connecting two vertices. These vertices don't have any specific features by themselves; they're just points or nodes. They can represent things like cities, concepts, or any other objects you want to connect in a network. When two vertices are connected by an edge, they're considered the endpoints of that edge, and you can say they're adjacent to each other.

(https://en.wikipedia.org/wiki/Vertex_(graph_theory))

Graph: A graph is a data structure that represents relationships between elements. It consists of vertices (or nodes) and edges that connect these vertices. The formal definition of a graph includes a set of vertices (V) and a set of edges (E), typically denoted as G(V, E).

(https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/)

Adjacency Matrix: An adjacency matrix is a way to represent a graph using a two-dimensional array of size N x N, where N is the number of nodes in the graph. The value adjMatrix[i][j] in this array represents the relationship or weight between node i and node j.

(https://www.thecshandbook.com/Adjacency_Matrix)

Adjacency List: An adjacency list is a memory-efficient representation for graphs, particularly well-suited to sparse graphs with relatively few edges compared to vertices. It associates each vertex with a list or array containing references to its neighboring vertices or edges. This representation offers several benefits, including easy traversal of neighbors, efficient memory usage, flexibility for

edge attributes, and suitability for directed graphs. It is commonly used in various applications, including social networks and recommendation systems.

(https://en.wikipedia.org/wiki/Adjacency_list)

BFS Traversal: The Breadth First Search (BFS) algorithm is a method for systematically exploring a graph data structure in search of a specific target node that satisfies certain criteria. It commences its search at the graph's root or initial node and proceeds by visiting and enqueuing all neighboring nodes at the same depth level. The algorithm then dequeues and visits nodes in a breadth-first manner, ensuring that all nodes at the current level are explored before moving to nodes at the subsequent depth level. This process continues until the target node is found, or the entire graph has been exhaustively traversed without success.

(https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/)

DFS Traversal: The Depth First Search (DFS) algorithm is a technique for systematically navigating a graph data structure in search of a specific target node that meets particular criteria. It begins its exploration at the root or starting node of the graph and follows a depth-first approach. DFS visits a node, explores as far down one branch as possible, then backtracks only when necessary. This process continues until the target node is found or until the entire graph is exhaustively traversed without success.

(https://www.programiz.com/dsa/graph-dfs)

Dijkstra Algorithm: Dijkstra's algorithm is a graph search algorithm that efficiently finds the shortest path from a specified source node to all other nodes in a weighted graph. It operates by iteratively selecting the node with the smallest tentative distance from the source and updates the distances to its neighboring nodes, effectively building a shortest-path tree. Dijkstra's algorithm ensures that it explores nodes in order of their distance from the source, guaranteeing the discovery of the shortest paths to all reachable nodes. This process continues until all nodes have been visited, resulting in a collection of shortest paths from the source to all other nodes in the graph.

(https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/)

Spanning Tree: In the field of graph theory, a spanning tree T of an undirected graph G is a subgraph that takes the form of a tree and includes all the vertices of G. It's worth emphasizing that, while there can be several different spanning trees for a graph, a fundamental prerequisite for the existence of a spanning tree is the graph's connectivity; a graph that isn't connected cannot possess a spanning tree.

(https://en.wikipedia.org/wiki/Spanning_tree#:~:text=In%20the%20mathematical%20field%20of,see%20about%20spanning%20forests%20below).)

Floyd-Wharshall Algorithm: Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm works for both the directed and undirected weighted graphs.

(https://www.programiz.com/dsa/floyd-warshall-algorithm)

Prim Algorithm: Prim's algorithm, a fundamental graph algorithm, is used to find the minimum spanning tree (MST) in a connected, undirected graph with weighted edges. It starts with an initial vertex and systematically adds edges of minimum weight, ensuring that the resulting tree remains connected and cycle-free. By iteratively growing the MST, Prim's algorithm guarantees a tree that spans all vertices with the least total edge weight, making it invaluable in applications like network design and infrastructure planning.

(https://www.javatpoint.com/prim-algorithm**)**

Kruskal Algorithm : The Kruskal algorithm is a fundamental graph algorithm that aims to find the minimum spanning tree (MST) in a connected, undirected graph with weighted edges. Unlike Prim's algorithm, Kruskal's algorithm operates by sorting all edges in the graph by their weights and then systematically adding the edges with the smallest weights as long as they do not create cycles in the evolving MST. This approach ensures that the resulting tree connects all vertices with the minimum possible total edge weight. Kruskal's algorithm is often implemented using a disjoint-set data structure to efficiently track and merge connected components. It is valuable in various applications such as network design, spanning tree construction, and optimization problems where minimizing the total cost is essential.

(https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/)

User Interface: The user interface (UI) is the point of human-computer interaction and communication in a device. This can include display screens, keyboards, a mouse, and the appearance of a desktop. It is also the way through which a user interacts with an application or a website.
(https://www.techtarget.com/searchapparchitecture/definition/user-interface-UI)

PHASE 3: SEARCH FOR CREATIVE SOLUTIONS

User interface: Since we need to implement a user interface we came out with 3 possible platforms and frameworks we could use:

JavaFX:
JavaFX is a versatile Java framework that can be used to create the user interface for "Blitz Bomb." It offers a straightforward way to design and implement a visually appealing game UI with interactive components. JavaFX's scene graph simplifies the management of UI elements and their layout. Additionally, we can leverage CSS styling to customize the appearance of the UI, giving the game a unique look and feel.

Web Environment:
Opting for a web-based user interface for "Blitz Bomb" opens up exciting possibilities. By utilizing Java-based web frameworks like Spring Boot, Java EE, or Play Framework, we can create a user interface accessible through web browsers on various devices. HTML and CSS provide the structure and styling for the UI, making it responsive and cross-platform. The addition of JavaScript enables interactivity, animations, and dynamic content, handling user

input and interactions in real-time. WebGL, a JavaScript API, is a great option for advanced graphics, ensuring a high-quality gaming experience through web browsers.

Other Java Frameworks:
Besides JavaFX and web environments, there are other Java frameworks that we considered for the "Blitz Bomb" user interface. LibGDX is a game development framework perfect for 2D and 3D games, offering tools for graphics rendering and user input. Java Swing, a traditional UI toolkit, can create a basic user interface for the game, while Java AWT, the Abstract Window Toolkit, provides another option for UI development.

Paths over Graph: We need to decide how we are going to traverse the graph to calculate how long it would take to traverse it on average.

Breadth-First Search (BFS):
BFS is a graph traversal algorithm that explores the graph level by level, starting from the source vertex.

Depth-First Search (DFS):
DFS is another graph traversal algorithm that explores one branch deeply before backtracking.

Power-ups: Which algorithm we are going to use to give kelvin the power up of knowing the shortest path between 2 nodes.

Dijkstra Algorithm:
Dijkstra's algorithm is used to find the shortest path from a node to another node by iteratively selecting nodes with the smallest tentative distance. In the game, it adds the possibility of giving the user a possibility of deciding a path without having to think. Giving him more options to get a better score.

Floyd-Warshall Algorithm:
The Floyd-Warshall algorithm is valuable when players need to find the shortest paths between all pairs of vertices in the graph. It offers a way of knowing what is the shortest path between 2 nodes therefore given the user the possibility of making faster moves.

Minimum Covering Tree - MST: Which algorithm are we going to use to decide the shortest path to traverse all the graphs and calculate on average how long we should give the player to finish the level.

Prim Algorithm:
Prim's algorithm builds the MST by adding edges of minimum weight while ensuring connectivity. This approach can be used to know what is the best way to traverse the whole graph ensuring that the player will explode all the bombs.

Kruskal Algorithm:
Kruskal's algorithm focuses on sorting edges by weight and adding them to the MST without forming cycles. This approach can be used to know what is the best way to traverse the whole graph ensuring that the player will explode all the bombs.

Calculate the Score:

Graph Traversal Efficiency Score:
In this option, we'll calculate a player's score based on how efficiently they traverse the game's graph. We will calculate the total time that it will take for someone to traverse the whole graph in an efficient way to determine if the user lost or not. If the user manages to complete the puzzle, his score will be the amount of time it took.
Bomb Activation Combo Score:
For this option, we'll introduce a combo system that rewards players for activating multiple bombs in a sequence. When players activate bombs without deviating from the path, they'll accumulate a combo multiplier. The longer the combo, the higher the score multiplier. However, if they deviate from the path or take too long between bomb activations, the combo is reset.
Graph Complexity Bonus Score:
In this option, we'll assign bonus points based on the complexity of the graph structure in each level. The more intricate the graph, with factors like the number of vertices, edge density, and challenging puzzles, the higher the potential score.

PHASE 4: TRANSITION FROM BRAINSTORMING TO PRELIMINARY DESIGNS

In the fourth phase of the engineering method, the process of discarding ideas that are not viable for solving the problem is crucial.

User Interface (UI):

In our careful assessment of UI options, we have chosen to discard the option of a **web-based user interface for "Blitz Bomb."** Additionally, we have decided not to explore other Java frameworks. The reason behind this decision stems from the intricate nature of web-based UI development, which presents exciting possibilities but, regrettably, exceeds the constraints of our project's time frame. Furthermore, introducing entirely new **frameworks** would necessitate substantial time investments from our team members, which is not conducive to our timely project completion objectives.

Scoring Mechanisms:

For scoring mechanisms, we have decided to eliminate the **Bomb Activation Combo Score** option. This choice has been made in recognition of the potential complexities it could introduce to the gameplay, without offering substantial improvement to it. Moreover, we have chosen to discard the exploration of the **graph complexity bonus score.** The reason for this

is because, in our intent to generate game maps in a random manner. This randomness introduces a challenge in determining the level of difficulty of a given map, rendering this particular scoring element less practical for our project.

The remaining ideas to be evaluated are:
-BFS,
-DFS,
-Dijkstra,
-Floyd-Warshall,
-Prim Algorithm
-Kruskal Algorithm.

PHASE 5: EVALUATION AND SELECTION OF THE BEST SOLUTION

Traverse the Graph to know how long it would take to traverse it all adding up all the weights between graphs.

Criterion A: Algorithm Complexity:

- [1] Linear O(n)
- [2] Logarithmic O(n log(n))
- [3] Logarithmic O(log(n))
- [4] Constant O(1)

Criterion B: Recursion:

- [1] NO
- [2] YES

Criterion C: Helper Data Structure

- [1] Queue
- [2] Stack

	Criterion A	**Criterion B**	**Criterion C**	**Total**
BFS	1	1	1	3
DFS	1	2	2	5

Selection:

Since the DFS is easier to implement thanks to recursion and uses less memory by implementing a stack we have decided to go with this option.

Adding power ups to the game

Criterion A: Simplicity to implement:

- **[1]** Hard
- [2] Medium
- [3] Easy

Criterion B: Efficiency in big Graphs:

- [1] O(n3)
- [2]O(n2)
- [3] Linear O(n)
- [4] Constant O(1)

Criterion C: Memory Usage:

- [1] O(n3)
- [2] O(n2)
- [3] O(n)

Criterion D: Gives shortest path between 2 nodes

- **[1] NO**
- **[2] YES**

	Criterion A	**Criterion B**	**Criterion C**	**Criterion D**	**Total**
Dijkstra	3	2	2	2	9
Floyd-Warshall	2	1	2	2	7

Selection:

After this comparison we have decided that implementing Dijkstra is the best option to give the player the power of knowing the shortest path between 2 nodes without including the bombs.

Algorithm to know the shortest path to traverse the whole graph with the goal of getting an approximation on how much time given the player.

Criterion A: Algorithm Complexity:

- [1] Linear O(n)
- [2] O(n2)
- [3] Logarithmic O(n log(n))
- [4] Constant O(1)

Criterion B: Can set where the algorithm starts.

- **[1] NO**
- **[2] YES**

Criterion C: Memory Usage:

- [1] O(n3)
- [2] O(n2)
- [3] O(n)

Criterion D: Runs better

- [1] small graphs
- [2] big graphs

	Criterion A	**Criterion B**	**Criterion C**	**Criterion D**	**Total**
Kruskal	3	1	3	1	8
Prim	2	2	3	2	9

Selection:

After this comparison, we have concluded that implementing the Prim algorithm is the best option to determine the shortest path to traverse the entire graph.

After all of this conclusion we decided we are going to use JavaFx for the UI, DFS to calculate an average, Dijkstra for the power Up and Prim to know what is the shortest way to traverse a graph.

PHASE 6: PREPARATION OF REPORTS AND SPECIFICATIONS

All other documents can be found in the DOCS folder of our project.