BOMBERMAN DOCUMENTATION

By:
Juan Camilo Muñoz
Juan David Acevedo
Manuel Cardona

PROBLEM

Bomberman is a classic Nintendo game that has seen several remakes over the years. In this integrative task, you are asked to develop your own version of the game while preserving the essence of the classic single-player version.

The game consists of three connected scenarios, and the player must traverse them while eliminating enemies in each one. To do this, the player has the ability to move up, down, left, and right across the map, constrained by walls, some of which are indestructible, and others that can be destroyed.

The player starts with only one bomb, but as they break destructible walls, they can pick up more bombs and acquire special abilities that modify their behavior. Additionally, the player's life status is displayed on the screen with three graphical elements (e.g., hearts), and they can lose a life if they are attacked by an enemy or caught in a bomb explosion.

The objective is to eliminate the enemies and be the last one standing, allowing the player to advance to the exit and win the game. In case of elimination, a Game Over screen will be displayed, and the player will have the opportunity to try again.

The game also includes enemies with basic movement routines that will attempt to eliminate the player when they spot them. Both the enemies and the player can be caught in a bomb explosion, resulting in the elimination of the enemy and the loss of a life for the player. The number of enemies is randomly generated to keep the game fun. The player can see the number of bombs they are equipped with and use them strategically to defeat the enemies. Game constraints include fixed map designs and the use of images or sprites for characters and map walls.

Note: Taken from the third integrative task in the Algorithms and Programming course:https://docs.google.com/document/d/1s_AX4SIW261CW7jWDRyuWMghspTFslC2/edit?usp=sharing&ouid=109415827520879394849&rtpof=true&sd=true

ENGINEERING METHOD

Phase 1: Problem identification

Identification of needs and symptoms:

In the initial phase of developing our Bomberman application, it is essential to clearly identify the needs and challenges we must address. Here, we define some of the specific needs and conditions under which we must solve these challenges:

- Develop a version of the Bomberman game with three connected scenarios, where the player eliminates enemies in each one and collects power-ups as they progress. The goal is to eliminate the enemies and reach the exit.
- Design a user interface for the player to interact with the game effectively.
- Implement player movement and game logic, including collision detection with walls and enemies.
- Manage the player's life system and the game ending logic (Game Over or victory).
- Allow enemies to chase the player and respond appropriately to bombs.

- Design a system for the random generation of enemies and the distribution of power-ups throughout the levels.
- Use images or sprites for characters and map walls.

Requirement Analysis

Client	**Algorithms and Programming, and Computing and Discrete Structures subject teachers**
User	The game player
Functional requirements	FR 1: Player Movement (Manuel Cardona) FR 2: Bomb Explosions (Juan David Acevedo) FR 3: Scenario Design (Juan Camilo Muñoz) FR 4: Player Collisions (Juan Camilo Muñoz) FR 5: Enemy Behavior(Juan Camilo Muñoz) FR 6: Collect Power-Ups (Juan David Acevedo) FR 7: Life System (Manuel Cardona) FR 8: Victory or Defeat Conditions (Juan David Acevedo)
Problem Context	For the third integrative task in Algorithms and Programming, the development of a version of the classic Nintendo game, Bomberman, is requested. This adaptation should consist of three connected scenarios where the player eliminates enemies, collects power-ups, and faces obstacles in the form of destructible and indestructible walls. The player has a set of functionalities, including movement in four directions, bomb placement, and life management. The main objective is to eliminate enemies and reach the exit to win the game while avoiding being caught in bomb explosions or attacked by enemies. The game version must incorporate elements such as enemy artificial intelligence, random generation of enemies and power-ups, and maintain a coherent graphic design with the classic game, using images or sprites for characters and the game environment.
Non-functional requirements	NFR 1: Graphic User Interface Development NFR 2: Use of Images or Sprites for the Player Character and Map Walls NFR 3: Random Enemy Generation

Identifier and Name	**FR 1 :** Player Movement		
Summary	The player can move up, down, left, and right across the map, constrained by walls. Some walls are indestructible, and others can be destroyed.		
Inputs	**Input Name**	**Type**	**Valid values condition**
	movementDirection	Button	Up, Down, Left, Right
Results	The player's movement is updated on the screen. If the player hits an indestructible wall, they cannot move in that direction. If the player		

	hits a destructible wall and has a bomb, they can choose to destroy the wall.		
Outputs	**Output Name**	**Type**	**Format**
	positionModification	N/A	N/A

Identifier and Name	**FR 2: Bomb explosions**		
Summary	In the game, bombs are objects with a predetermined time of explosion. The number of bombs the player can place will increase as they collect "abilities" during the game, with each collected "ability" increasing the number of bombs by one. The blast radius of a bomb will also vary based on the total number of "abilities" collected by the player. As the player accumulates more "abilities," the blast radius will be greater. It's important to note that the explosion of a bomb will disperse in the four directions: up, down, left, and right, unless there are obstacles that block any of these directions. The dispersion will occur evenly in all available directions from the bomb's point of origin. Additionally, in the game scenario, the player will be able to drop bombs by pressing a specific key.		
Inputs	**Input Name**	**Type**	**Valid values condition**
	Button	Event	NA
Results	After a bomb is thrown in the game, an explosion will occur that will affect enemies or the player, depending on their location within the bomb's blast radius. The explosion will have an impact on all entities, whether they are enemies or the player, that are within the range of the bomb.		
Outputs	**Output Name**	**Type**	**Format**
	Explosión	NA	NA

Identifier and Name	**FR 3: Scenario Design**
Summary	The scenarios are the areas where the player moves, places bombs, and battles enemies, making them an interactive area with the user. The scenario design should include the placement of obstacles, destructible walls, power-ups, and enemies. The game consists of 3 connected scenarios. From one scenario, the player can move to another using access points. This way, the player starts in scenario 1 and gradually progresses through the other two scenarios until they

<table>
<tr><td></td><td colspan="3">win or lose the game. At the beginning of the game, all three access points are blocked.

Key Points to Consider:
- The number of enemies in each scenario is randomly generated within a range we set.
- There are no specific guidelines on how the scenario should be designed, so the number of walls can be fixed or generated randomly. The same applies to power-ups.
- The scenarios do not need to be randomly generated.
- While it is assumed there should be a start menu, it's not explicitly mentioned as a requirement, so the game can start, and consequently, the scenario, as soon as the program launches.</td></tr>
<tr><td rowspan="2">Inputs</td><td>Input Name</td><td>Type</td><td>Valid values condition</td></tr>
<tr><td>StartGameButton</td><td>Button</td><td>The player must click it, in order to start the game and generate the maps.</td></tr>
<tr><td>Results</td><td colspan="3">The game environment is created, including the elements of the scenario as mentioned before. The player starts at the beginning of the map, and the enemies are placed in random positions, except near or on the player.</td></tr>
<tr><td rowspan="2">Outputs</td><td>Output Name</td><td>Type</td><td>Format</td></tr>
<tr><td>Map</td><td>Scene</td><td>NA</td></tr>
</table>

<table>
<tr><td>Identifier and Name</td><td colspan="3">FR 4: Player Collisions</td></tr>
<tr><td>Summary</td><td colspan="3">Collisions refer to the event in which two or more objects in the game world interact or overlap in some way. These interactions can have various implications in the game and can trigger specific events according to the game's rules.

The system must detect and handle collisions that the player has with game objects. These objects include bombs, power-ups, indestructible walls, and destructible walls.</td></tr>
<tr><td rowspan="2">Inputs</td><td>Input Name</td><td>Type</td><td>Valid values condition</td></tr>
<tr><td>movementButton</td><td>Button</td><td>NA</td></tr>
<tr><td>Results</td><td colspan="3">The result of object collision detection is to determine whether a collision has occurred between game elements. This includes identifying collisions between:

- Player Collision with Walls: Detect if the player is in a position that collides with indestructible or destructible walls, limiting their movement.
- Player Collision with Enemies: Determine if the player is reached by an enemy, which can result in the player taking damage.</td></tr>
</table>

<table>
<tr><td></td><td colspan="3">- Player Collision with Bombs: Determine if the player is in a position that collides with a bomb that has not yet exploded, limiting their movement.
- Player Collision with Explosions: Determine if the explosion from a bomb reaches and affects the player, potentially causing them to take damage.
- Player Collision with Power-Ups: Detect if the player moves to a position that contains a power-up, allowing the player to collect it and gain a benefit.</td></tr>
<tr><td rowspan="4">Outputs</td><td>Output Name</td><td>Type</td><td>Format</td></tr>
<tr><td>movement Restrictions</td><td>NA</td><td>NA</td></tr>
<tr><td>damage</td><td>NA</td><td>NA</td></tr>
<tr><td>pickingUpgrades</td><td>NA</td><td>NA</td></tr>
</table>

<table>
<tr><td>Identifier and Name</td><td colspan="3">FR 5: Enemy Behavior</td></tr>
<tr><td>Summary</td><td colspan="3">Enemies are non-player characters that move throughout the scenario and have the primary goal of defeating the player. These enemies typically have basic movement routines and can chase the player when they see them. They can also be caught in bomb explosions and, as a result, be defeated.</td></tr>
<tr><td rowspan="5">Inputs</td><td>Input Name</td><td>Type</td><td>Valid values condition</td></tr>
<tr><td>bombPositionX</td><td>int</td><td>NA</td></tr>
<tr><td>bombPositionY</td><td>int</td><td>NA</td></tr>
<tr><td>platerPositionX</td><td>int</td><td>NA</td></tr>
<tr><td>playerPositionY</td><td>int</td><td>NA</td></tr>
<tr><td>Results</td><td colspan="3">The result of enemy behavior is their response to the player's actions and their movement throughout the scenario. This includes:
- Player Pursuit: Enemies should be programmed to move towards the player when they spot them, following a route or pursuit strategy.
- Interaction with Walls and Obstacles: Enemies should be able to navigate obstacles and navigate around walls to effectively reach the player.
- Inflict Damage: When enemies reach the player and generate a collision, they should reduce the player's life by one heart.
- Receive Damage from Explosions: When an enemy is caught in a bomb explosion, the enemy is eliminated from the game.</td></tr>
<tr><td rowspan="3">Outputs</td><td>Output Name</td><td>Type</td><td>Format</td></tr>
<tr><td>recibeDamage</td><td>NA</td><td>NA</td></tr>
<tr><td>inflictDamage</td><td>NA</td><td>NA</td></tr>
</table>

	enemyMovement	NA	NA

Identifier and Name	**FR 6 :** Collect Power-Ups		
Summary	In the game, there will be various "abilities" that alter the behavior of the player and the bombs. These abilities can appear either by being dropped by defeated enemies or by destroying structures. The player will be able to acquire them by passing over them.		
Inputs	**Input Name**	**Type**	**Valid values condition**
	movementButton	Button	N/A
Results	Once the player collects an ability, their in-game behavior will be modified in accordance with the specific nature of the acquired ability.		
Outputs	**Output Name**	**Type**	**Format**
	modification	N/A	N/A

Identifier and Name	**FR 7: Life System**		
Summary	The player's life status is displayed on the screen with three graphical elements (e.g., hearts). A life can be lost if the player is attacked by an enemy or caught in a bomb explosion.		
Inputs	**Input Name**	**Type**	**Valid values condition**
	enemyAttack	NA	NA
	bombExplosion	NA	NA
Results	If the player is attacked by an enemy or caught in a bomb explosion, the system will decrease the life count by one. If all lives are lost, a Game Over screen will be displayed.		
Outputs	**Output Name**	**Type**	**Format**
	lifeCount	NA	NA
	gameOver	Boolean	NA

Identifier and Name	**FR 8: Victory or defeat conditions**		
Summary	Once the player defeats all the enemies present in the stage, a designated door at a specific point in the stage will open, allowing the player to advance to the next level		
Inputs	**Input Name**	**Type**	**Valid values condition**
	movementButton	Button	N/A

Results	Upon successfully completing the third level and crossing the corresponding door, the system will display a victory message, indicating that the player has won the game. If the player loses all three lives in any stage, a defeat message will be shown, indicating that the player has lost the game.		
Outputs	**Output Name**	**Type**	**Format**
	victoryMessage	String	Screen Message: "You won"
	defeatMessage	String	Screen Message: "You lose"

Phase 2: Gathering the necessary information

Main Mechanics of the Original Game

In order to gain a broader understanding of what is required in the task, we have focused on gathering relevant information and necessary data to address the development of a game based on Bomberman. To achieve this, we conducted a detailed investigation of the key mechanics and functionalities from the original Bomberman game for the Nintendo NES (1983). Below is a detailed description of these mechanics, highlighting the essential aspects that shape the Bomberman gaming experience:

- **Perspective and Scenario:** The game is set in a two-dimensional environment with a top-down view. The scenario consists of a gridded maze with walls, some destructible blocks, and indestructible blocks. The player controls Bomberman, a character that can move in four directions: up, down, left, and right.

- **Bomb Placement:** The central mechanic of Bomberman is the player's ability to plant bombs. The player can place a bomb in their current location, and these bombs explode in a cross-shaped explosion in four directions. The player starts with a single bomb but can collect power-ups throughout the game to increase their maximum quantity.

- **Wall Destruction and Power-Up Collection:** The player can use bombs to destroy destructible walls in the scenario. Behind some of these walls, power-ups are hidden, enhancing Bomberman's capabilities and making him more effective in destroying enemies and obstacles, such as more bombs or increased explosion range.

- **Enemies:** In the scenario, there are enemies that move predictably and pursue the player when they are in their line of sight. These enemies are eliminated by bomb explosions, and by doing so, some of them can drop power-ups. The main objective of the game is to eliminate all the enemies in the scenario to access the exit and advance to the next level. Once all enemies are defeated, and the player reaches the exit, the level is completed, and the game proceeds to the next one with increased difficulty.

- **Lives and Continuation**: Bomberman starts with a limited number of lives but gains more as they progress through the scenarios. Losing a life occurs if Bomberman is caught in a bomb explosion or captured by an enemy. The game ends when all of the player's lives are depleted, displaying a Game Over screen. However, the player has the opportunity to continue from the last completed level, allowing for some degree of recovery.

This information has been taken from:
https://www.miladonintendo.cl/analisis/bomberman-nes-1987/
https://www.youtube.com/watch?v=bSGalMTxuZA
Bomberman (videojuego) - Wikipedia, la enciclopedia libre
Bomberman - Wikipedia, la enciclopedia libre

Graph Theory
The use of graphs in video game development is a strategic choice due to the ability to model relationships and connections between game elements efficiently. Graphs provide a visual and structural representation of these elements and allow the application of algorithms to solve specific in-game problems. In Bomberman, graphs could be used to model the game map, routes for enemy "artificial intelligence", explosion management and more.

Graphs:
A graph is a mathematical structure consisting of nodes (vertices) and edges (edges) connecting these nodes. Each edge represents a relationship between two nodes, and graphs can be directed or undirected, weighted or unweighted, depending on the nature of the problem. In our context, nodes can represent locations within the scenario, characters, game elements, while edges represent connections or relationships between them.

Graphs Path:

1. BFS (Breadth-First Search): This algorithm is used to explore or search in width through a network. It is ideal for finding the shortest path or the fewest number of steps needed to get from a starting point to a destination point. In Bomberman, BFS can be used to find the shortest route for a character to a safe exit after an explosion.

2. DFS (Depth-First Search): DFS is an algorithm used to explore or deep search a network. It is useful for exploring all branches of a network and can be used to keep track of all locations accessible from a starting point. In Bomberman, DFS could be useful for exploring adjacent areas and determining possible movement paths for a character.

Minimum Weight Paths:

3. Dijkstra: Dijkstra's algorithm is used to find the shortest path between two nodes in a weighted graph with non-negative edges. In the context of Bomberman, Dijkstra could be useful for calculating the shortest distance from the player's position to a power or an important element on the map.

4. Floyd-Warshall: This algorithm is used to find all shortest paths between all pairs of nodes in a weighted graph, which is useful in games where optimal paths between multiple points need to be computed, such as planning the movement of multiple enemies in Bomberman.

Minimum Covering Tree -MST-:

5. Prim: Prim's algorithm is used to find a minimum spanning tree in an undirected weighted graph. In the context of Bomberman, it could be applied to model the expansion of an

explosion from a bomb as a minimum covering tree to calculate which obstacles it destroys or also for map generation. For this ultimate purpose, the process is as follows:
Start with a grid full of walls. (taken from: https://en.wikipedia.org/wiki/Maze_generation_algorithm)

1. Pick a cell, mark it as part of the maze. Add the walls of the cell to the wall list.
2. While there are walls in the list:
    1. Pick a random wall from the list. If only one of the cells that the wall divides is visited, then:
        1. Make the wall a passage and mark the unvisited cell as part of the maze.
        2. Add the neighboring walls of the cell to the wall list.
    2. Remove the wall from the list.

6. Kruskal: Kruskal is another algorithm for finding a minimum covering tree in an undirected weighted graph. In Bomberman, it could be used to model the progressive destruction of obstacles on the map as explosions propagate.

This information has been taken from:
https://www.youtube.com/watch?v=LFKZLXVO-Dg
https://www.youtube.com/watch?v=PMMc4VsIacU
https://www.youtube.com/watch?v=xlVX7dXLS64
Sesiones 26-32 de clases del curso de Computacion y Estructuras Discrertas

Phase 3: Search for creative solutions

In this third phase of the engineering process to develop our version of the Bomberman game, we have made the decision to use graph models as the central representation of the game. This choice allows us to effectively address the complex interactions and mechanics of the game, providing a solid framework for design and implementation. Throughout this phase, we will focus on the possible graph theory algorithms that may be most relevant in the game's design.

To provide context, in the Bomberman game, the character (player) has the ability to collect three types of power-ups: one that increases the explosion range (cumulative) and that increases the number of bombs the player can place (cumulative). Additionally, the player's life status is represented by three hearts, and losing a life occurs when the player is caught in a bomb explosion or attacked by an enemy.

Idea 1: Challenge Mode with Shortest Path (Dijkstra) and Minimum Spanning Tree (Prim)

In this version of the Bomberman game, a challenge mode is introduced where the player must face complex mazes that require strategic thinking. The application of graph algorithms such as Dijkstra and Prim plays a crucial role in the game's logic. Here are the mechanics and how these algorithms are used:

Mechanics:

- **Maze Generation:** The scenario is generated as a maze with multiple paths, obstacles, enemies, and an exit. The maze is procedurally generated in each level of the challenge.
- **Bomb Placement**: Bomberman retains the ability to plant bombs, which are essential for eliminating obstacles and enemies on the way to the exit.

- **Main Objective**: The player's goal is to reach the maze's exit, and to do so, they must find the shortest and safest path while avoiding enemies and obstacles.

Use of Algorithms:

- **Dijkstra for Shortest Paths:** The Dijkstra algorithm is used to calculate the shortest path from Bomberman's current position to the exit. This takes into account the length of the routes and the enemy locations. The result serves as a guide for the player to reach the exit.
- **Minimum Spanning Tree (Prim):** The Prim algorithm is employed to generate a minimum spanning tree within the maze. This tree connects key areas of the scenario and provides strategic routes for the player. Bomberman can use these routes to avoid obstacles and enemies while searching for the exit.

Game Flow:

- The player starts in a random location within the maze and must use the information provided by the Dijkstra algorithm to determine the shortest path to the exit.
- During their search, Bomberman can plant bombs to eliminate obstacles and enemies blocking their path.
- The player should consider the routes generated by the Prim algorithm while exploring the maze. These routes provide shortcuts and strategic options to avoid hazards and efficiently reach the exit.
- The challenge is completed when the player successfully reaches the maze's exit, taking into account the recommendations of the algorithms and overcoming obstacles along the way.

Idea 2: Directed Multigraph with Graph Traversals (BFS) and Shortest Paths (Floyd-Warshall)

In this version of the Bomberman game, a directed multigraph is introduced, allowing Bomberman to move in multiple directions. Graph algorithms such as BFS and Floyd-Warshall are used to guide the player's movement and strategy. Here are the mechanics and how these algorithms are used:

Mechanics:

- **Multi-directional Movement:** In this game, Bomberman has the ability to move in multiple directions, including up, down, left, and right.
- **Bomb Placement:** Bomberman retains the ability to plant bombs, which remain essential for eliminating obstacles and enemies on the way to the goal.
- **Main Objective:** The player's goal is to navigate the maze efficiently, eliminating enemies, and reaching the exit.

Use of Algorithms:

- **Graph Traversals (BFS):** The BFS algorithm is used to calculate possible movement routes from Bomberman's current position. This takes into account allowed directions and multigraph restrictions. The calculated routes are shown to the player as movement options.
- **Shortest Paths (Floyd-Warshall):** The Floyd-Warshall algorithm is employed to calculate the shortest paths between all nodes in the multigraph. This allows the player to make strategic decisions based on minimum distances to objectives or to avoid enemies.

Game Flow:

- The player starts at a random location within the multigraph and must use the routes generated by the BFS algorithm to determine movement options.
- During their exploration, Bomberman can plant bombs to eliminate obstacles and enemies blocking their path.
- The player must consider the minimum distances calculated by the Floyd-Warshall algorithm to make strategic decisions and avoid enemies, optimizing their route to the goal.
- The challenge is completed when the player successfully reaches the maze's exit, having used the information from the algorithms and overcome obstacles.

Idea 3: Timed Challenge with Shortest Paths (Dijkstra)

In this version of the game, a time element is introduced, and the player must complete levels within a limited time. The Shortest Paths algorithm (Dijkstra) is used to calculate the shortest path to the goal, considering the time constraint.

Mechanics:

- **Timed Challenge**: The player must complete each level within a specified time, adding pressure and excitement to the game.
- **Bomb Placement:** Bomberman retains the ability to plant bombs to open paths and eliminate enemies.
- **Main Objective:** The player's goal is to reach the level's exit before time runs out.

Use of the Algorithm:

- The Dijkstra algorithm is employed to calculate the shortest path from Bomberman's current position to the exit, taking into account the limited time. This helps the player make quick and efficient decisions.

Game Flow:

- The player starts in a level and must navigate swiftly to reach the exit before time runs out.
- Strategic bomb placement is essential to open paths and eliminate obstacles while rushing toward the exit.
- The Dijkstra algorithm provides guidance to the player about the shortest route, allowing for quick and efficient decisions to reach the exit on time.
- The challenge is completed if the player manages to reach the exit before time runs out.

Idea 4: Player Search with BFS and Map Autogeneration using Prim

The game is set in a maze consisting of a minimum of 50 cells (vertices) and corridors (edges). Bomberman retains the ability to plant bombs and eliminate obstacles and enemies. The main objective remains to eliminate all enemies and reach the exit.

Implementation of Graph Algorithms:

- **Graph Traversal (BFS):** Implements the BFS algorithm for enemies to efficiently search for the player.
- **Minimum Spanning Tree (Prim)**: Uses the Prim algorithm to generate game levels. The maze is created, ensuring that all cells are connected in a Minimum Spanning Tree, ensuring that the level is traversable and efficiently connected.

Game Mechanics:

- Bomberman has the ability to move in multiple directions, including up, down, left, and right.
- Bomb placement to eliminate obstacles and enemies remains essential.
- Enemies search for the player using BFS to take efficient routes.

Phase 4: Transition from the formulation of ideas to designs preliminary

Here are the pros and cons of each idea:

Idea 1: Challenge Mode with Minimum Weight Paths (Dijkstra) and Minimum Spanning Tree (Prim):

Pros:

- The combination of Dijkstra and Prim adds depth and complexity to the game.
- Allows for procedural maze generation, ensuring variety in each level.

Cons:

- The implementation of complex algorithms can increase the learning curve and difficulty of the game, which may not be suitable for all players.
- Procedural maze generation can be an unnecessary challenge in terms of design and performance. It's not a mandatory requirement.

Idea 2: Directed Multigraph with Graph Traversal (BFS) and Minimum Weight Paths (Floyd-Warshall):

Pros:

- Introduces a new multidirectional approach that adds variety and strategy to the game.
- The use of algorithms for informed decision-making provides an additional challenge.
- Players must carefully plan their route, adding depth to the game.

Cons:

- The introduction of multidirectional movement and algorithms can increase complexity and require a steeper learning curve.
- Managing a directed multigraph may be more complicated from a design and implementation perspective.

Idea 3: Timed Challenge with Minimum Weight Paths (Dijkstra):

Pros:

- Adds a time element that adds excitement and pressure to the game.
- The implementation of Dijkstra for optimal routes within a time limit provides a challenging experience.

Cons:

- Time management can be stressful for some players.
- The Floyd-Warshall algorithm is used to find the minimum path in the graph, which can be repetitive considering BFS does the same.

Idea 4: Player Search by BFS and Map Autogeneration with Prim:

Pros:

- Introduces a player search approach by enemies, adding a strategic challenge.
- Map autogeneration with Prim ensures connected and playable levels.
- Combines classic Bomberman mechanics with innovative design elements.

Cons:

- Player search by enemies could result in a more challenging game.
- Implementing map autogeneration may require additional development effort.

Since all ideas have their merits, **Idea 3: Timed Challenge with Minimum Weight Paths (Dijkstra)** could be considered for elimination since the inclusion of the time factor and time management is not required, further complicating software design. The other ideas offer interesting strategic and design approaches that maintain the essence of Bomberman.

Phase 5: Evaluation and selection of the best solution

In the phase of selecting the best idea for our Bomberman game, we have decided to combine ideas 2 and 4, since they share common points and offer a broader and more strategic gaming experience. Both ideas incorporate elements of multi-directional movement, planting bombs, efficiently searching for enemies, and generating efficiently connected game levels. By uniting Bomberman's multi-directional movement capability with the implementation of algorithms like BFS and Prim, we created a game where players must navigate connected mazes efficiently, confront strategically searching enemies, and intelligently use bombs to overcome obstacles. It is worth mentioning that the implementation of the Floyd-Warshall algorithm by idea 2 is ruled out.

We have discarded idea 3, which is based on the Floyd-Warshall algorithm, since the BFS implementation in ideas 2 and 4 covers efficient path finding without adding Floyd-Warshall complexity.

We have also discarded idea 1, as it is quite similar to idea 4 in terms of using efficient pathfinding and maze generation algorithms, but idea 4 offers a broader proposal.

ADT DESIGN

Graph ADT			
G = (V, E), where V is the set of vertices and E is the set of edges.			
{inv: $\forall$ (Vi , Vj) $\rightarrow$ (i $\neq$ j) = An already existing vertex can't be added.}			
Operation	**Type**	**Input**	**Output**
Graph	*Constructor*	-	Graph
bfs	*analyzer*	Graph x key	Graph
dfs	*analyzer*	Graph	Graph
dijkstra	*Analyzer*	Graph x key x key	ArrayList<Integer>
floydWarshall	*Analyzer*	Graph	int[][]
prim	*analyzer*	Graph	ArrayList<Edges>
kruskal	*analyzer*	Graph	Arraylist<Edges>
adjacent	*analyzer*	Graph x key x key	boolean
deleteEdge	*modifier*	Graph x key x key	Graph
insertEdge	*modifier*	Graph x key x key x weight	Graph
deleteVertex	*modifier*	Graph x key	Graph
insertVertex	*modifier*	Graph x key x value	Graph

Graph() "Creates a new graph" {pre: TRUE} {pos: Creates a graph}
bfs(G, k) "Explores the graph G starting on the vertex with key k and carries on with all its neighbors" {pre: k1 must belong to a vertex in the set of vertexes of the graph G} {post: All nodes reachable from the source vertex}
dfs(G) "Explores all the graph G starting in the first vertex to determine their vertexes distances" {pre: TRUE} {post: All vertexes visited during the DFS traversal}

dijkstra(G, k1) “Returns the shortest path between the vertexes with key k1 and all the others vertexes”   {pre: G must be an undirected or directed weighted graph without negative cycles}   {post: Shorter distances from one source node to all are returned}
floydWarshall(G) “Returns the shortest path between all the pair of vertexes”   {pre: G must be a weighted graph without negative cycles.}   {post: All shortest distances between all pairs of nodes are returned}
prim(G) “Creates a minimal spanning tree from an initial node.”   {pre: G must be undirected and connected graph with non-negative edge weights}   {post: get a minimum spanning tree connecting all vertexes of the graph G}

kruskal(G) “Creates a minimal spanning tree with no cycles and minimal weight.” {pre: G must be undirected and connected graph with non-negative edge weights} {post: get a minimum spanning tree of the graph G that connects vertices without cycles and with the minimum weight}
adjacent (G, k1, k2) **“**Returns true if vertexes with keys k1 and k2 form an edge” {pre: There must be an edge between the vertexes with keys k1 and k2} {pos: true if vertexes with keys k1 and k2 form an edge. False otherwise}
deleteEdge(G, k1, k2) “Deletes the edge between the vertexes with keys k1 and k2 of the graph G” {pre: There must be an edge between the vertexes with keys k1 and k2} {pos: The edge is removed from the graph G}

insertEdge(G, k1, k2, weight) “Adds an edge between the vertexes with keys k1 and k2 with the specified weight to the graph G”   {pre: k1 and k2 keys must belong to vertexes in the set of vertexes of the graph G}   {pos: A weighted edge connecting the vertexes with keys k1 and k2 has been created in the graph G}
deleteVertex(G, k) “Deletes a vertex with the specified key of the graph G”   {pre: k must be a key of a vertex in the set of vertices of the graph G}   {pos: The vertex is removed from the graph G}
insertVertex(G, k, value) “Adds a new vertex in the graph G”   {pre: G = {} ∧ the new vertex must not belong to the vertex set}   {post: The vertex has been added to the graph G

UNIT TEST DESIGN

Class	Name	Scenario
RandomMapGeneratorTest	setup1	The generated map is not null
RandomMapGeneratorTest	setup2	The generated map has the correct size
RandomMapGeneratorTest	setupGeneratedMapBoundaries	The generated map has the correct boundaries
GraphsTest	setup1	The graph is empty
GraphsTest	setup2	The graph is not empty

`GraphsTest:`

Test Objective	Class and Method	Scenario	Input Values	Expected Result
Test addVertex in an empty graph	GraphsTest.testAddVertexEmptyGraph	setup1	Key: 1, Value: "A"	Should return true, and the value associated with key 1 should be "A".
Test addVertex in a non-empty graph	GraphsTest.testAddVertexNonEmptyGraph	setup2	Key: 1, Value: "C"	Should return false since the vertex with key 1 already exists. Should add the vertex with key 3 and value "C".

Test removeVertex in an empty graph	GraphsTest.testRemoveVertexEmptyGraph	setup1	Key: 1	Should return false since the vertex with key 1 does not exist.
Test removeVertex in a non-empty graph	GraphsTest.testRemoveVertexNonEmptyGraph	setup2	Key: 1	Should return true, and the vertex with key 1 should be null. Also, the edge between 1 and 2 should be removed.
Test addEdge in an empty graph	GraphsTest.testAddEdgeEmptyGraph	setup1	Keys: 1, 2; Weight: 5	Should return false since the vertices with keys 1 and 2 do not exist.
Test addEdge in a non-empty graph	GraphsTest.testAddEdgeNonEmptyGraph	setup2	Keys: 2, 1; Weight: 4	Should return true. The edge between 2 and 1 should exist, and its weight should be 4.
Test removeEdge in an empty graph	GraphsTest.testRemoveEdgeEmptyGraph	setup1	Keys: 1, 2	Should return false since the vertices with keys 1 and 2 do not exist.
Test removeEdge in a non-empty graph	GraphsTest.testRemoveEdgeNonEmptyGraph	setup2	Keys: 1, 2	Should return true. The edge between 1 and 2 should be removed.
Test hasEdge in a non-empty graph	GraphsTest.testHasEdgeNonEmptyGraph	setup2	Keys: 1, 2	Should return true. The edge between 1 and 2 exists.
Test hasEdge in an empty graph	GraphsTest.testHasEdgeEmptyGraph	setup1	Keys: 1, 2	Should return false since the vertices with keys 1 and 2 do not exist.

Test bfs in an empty graph	GraphsTest.testBFSEmptyGraph	setup1	Key: 1	Should return null since the vertex with key 1 does not exist.
Test bfs in a non-empty graph	GraphsTest.testBFSNonEmptyGraph	setup2	Key: 1	Should return a queue with size 3, and the value of the first element should be "A".
Test dfs in an empty graph	GraphsTest.testDFSEmptyGraph	setup1	Key: 1	Should not throw exceptions since the vertex with key 1 does not exist.
Test dfs in a non-empty graph	GraphsTest.testDFSNonEmptyGraph	setup2	Key: 1	Should not throw exceptions. The color of vertices 1 and 2 should be Color.BLACK.
Test getVertices in an empty graph	GraphsTest.testGetVerticesEmptyGraph	setup1		Should return an empty list since the graph is empty.
Test getVertices in a non-empty graph	GraphsTest.testGetVerticesNonEmptyGraph	setup2		Should return a list with two elements, "A" and "B".
Test getVertex in an empty graph	GraphsTest.testGetVertex	setup1	Key: 1	Should return null since the vertex with key 1 does not exist.
Test getVertex in a non-empty graph	GraphsTest.testGetVertexNonEmptyGraph	setup2	Key: 1	Should return the vertex with key 1 and value "A".

RandomMapGeneratorTest：

Test Objective	Class and Method	Scenario	Input Values	Expected Result
Test getGeneratedMap not null	RandomMapGeneratorTest.testGeneratedMapNotNull	setup1		Should return a non-null generated map.
Test generated map size	RandomMapGeneratorTest.testGeneratedMapSize	setup1		The size of the generated map should be 10x10.
Test generated map boundaries	RandomMapGeneratorTest.testGeneratedMapBoundaries	setup1		The boundaries of the generated map should be TileState.BLOCKED.