INTEGRATIVE TASK 2

DAVIDE FLAMINI CAZARAN - A00381665 - SYSTEMS ENGINEERING

NICOLÁS CUÉLLAR MOLINA - A00394970 - SYSTEMS ENGINEERING

ANDRES CABEZAS GUERRERO - A00394772 - SYSTEMS ENGINEERING

ICESI UNIVERSITY

FACULTY OF ENGINEERING

COMPUTING AND DISCRETE STRUCTURES I

PROBLEM SPECIFICATION TABLE

CLIENT	Territory Conquest Game
USER	Player
FUNCTIONAL REQUIREMENTS	● R1: Add territories to the game ● R2: Establish connections between adjacent territories ● R3: Allow players to conquer adjacent territories ● R4: Calculate the score of each player based on the territories conquered ● R5: Use graph traversal algorithms (BFSo DFS) to determine the areas conquered by each player
CONTEXT OF THE PROBLEM	The creation of a strategy game based on the conquest of territories is required. Additionally, there must be an efficient system for managing the territory conquest process in the game. Players must conquer adjacent territories, represented as vertices in a graph, where the edges represent the connections between the territories. The objective is to determine the areas conquered by each player and calculate the corresponding score.
NON-FUNCTIONAL REQUIREMENTS	- RN1: Representation of Graphs, by means of own and implemented codes, the problem must be solved. - RN2: Intuitive user interface - RN3: The project must be uploaded to the GitHub platform and must have changes that allow the evolution of the project to be tracked.

REQUIREMENT SPECIFICATIONS

Name or identifier	R1: Add Territory		
Summary	The system should allow the addition of a territory to the game, along with its corresponding connections.		
Inputs	**input name**	**Datatype**	**Selection or repetition condition**
	territoryName	String	N/A
	adjacentTerritories	Array/List of Strings	N/A
General activities necessary to obtain the results	1. Prompt the user to enter the name of the territory to be added. 2. Store the entered territory name in the variable territoryName. 3. Prompt the user to enter the names of the adjacent territories. 4. Store the entered adjacent territories in the variable adjacentTerritories. 5. Create a new territory object using the provided information. 6. Add the new territory to the graph and establish the corresponding connections with the adjacent territories.		
Result or postcondition	The new territory is added to the game with its connections to the adjacent territories.		
Outputs	**output name**	**Datatype**	**Selection or repetition condition**
	confirmation	String	N/A

Name or identifier	R2: Establish connections between adjacent territories		
Summary	The system must allow establishing connections with weight between adjacent territories in the territory conquest game, where the weight represents the military power or strength of the connection.		
Inputs	**input name**	**Datatype**	**Selection or repetition condition**
	territorySource	String	N/A
	territoryDestination	String	N/A
	edgeWeigth	int	N/A
General activities necessary to obtain the results	1. Check if the origin and destination territories exist in the game. 2. Verify if the territories of origin and destination are different. 3. Check if the territories are adjacent to each other. 4. Check if there is already a connection between the territories. 5. Establish the connection with weight between the territories.		
Result or postcondition	The connection with weight between the adjacent territories is established, where the weight represents the military power or strength of the connection.		
Outputs	**output name**	**Datatype**	**Selection or repetition condition**
	N/A	void	N/A

Name or identifier	R3: Allow players to conquer adjacent territories		
Summary	The system must allow players to conquer Adjacent territories in the strategy game based on the conquest of territories. This implies that a player can select their own territory and attack an adjacent territory controlled by another player, with the aim of conquering it.		
Inputs	**input name**	**Datatype**	**Selection or repetition condition**
	attackingTerritory	String	N/A
	defendingTerritory	String	N/A
General activities necessary to obtain the results	1. Verify the validity of the territory of origin selected by the player. 2. Verify the validity of the target territory selected by the player. 3. Check if the source territory and the target territory are connected by an edge in the graph. 4. Calculate the probability of success of the attack based on the military power of the territories involved. 5. Update game settings in case the attack is successful. 6. Do not make any changes to the game settings if the attack fails.		
Result or postcondition	The player's conquered territories are updated in the game settings and a confirmation message if the attack was successful.		
Outputs	**output name**	**Datatype**	**Selection or repetition condition**
	confirmation	String	N/A

Name or identifier	R4: Calculate the score of each player based on the territories conquered		
Summary	In this requirement, it is sought to calculate the score of each player based on the territories conquered in the strategy game based on the conquest of territories.		
Inputs	**input name**	**Datatype**	**Selection or repetition condition**
	playerTerritories	ArrayList	N/A
	edgeWeigths	ArrayList	N/A
General activities necessary to obtain the results	1. Calculate the size of the area conquered by the player based on the conquered territories. 2. Calculate the player's score based on the size of the conquered area. 3. Consider the weight of the edges that connect the territories within the conquered area to adjust the score, if necessary.		
Result or postcondition	Each player's score is calculated and updated in the game settings by displaying each player's score.		
Outputs	**output name**	**Datatype**	**Selection or repetition condition**
	playersScore	String	N/A

Name or identifier	R5: Use graph traversal algorithms (BFSo DFS) to determine the areas conquered by each player		
Summary	In this requirement, it is sought to use graph traversal algorithms, such as BFS (Breadth-First Search) or DFS (Depth-First Search), to determine the areas conquered by each player in the strategy game based on the conquest of territories.		
Inputs	**input name**	**Datatype**	**Selection or repetition condition**
	player	Player	N/A
	playerTerritory	Territory	N/A
General activities necessary to obtain the results	1. Select a territory controlled by the player as the starting point. 2. Initialize an empty set or list to store the territories conquered by the player. 3. Take a walk over the graph using BFS and DFS, starting from the starting territory. 4. During the tour, mark each territory visited as part of the area conquered by the player and add it to the set or list of conquered territories. 5. Repeat this process for each territory controlled by the player. 6. Save the set or list of territories conquered by the player.		
Result or postcondition	The areas conquered by each player are determined using graph traversal algorithms.		
Outputs	**output name**	**Datatype**	**Selection or repetition condition**
	confirmation	String	

ENGINEERING DESIGN PROCESS

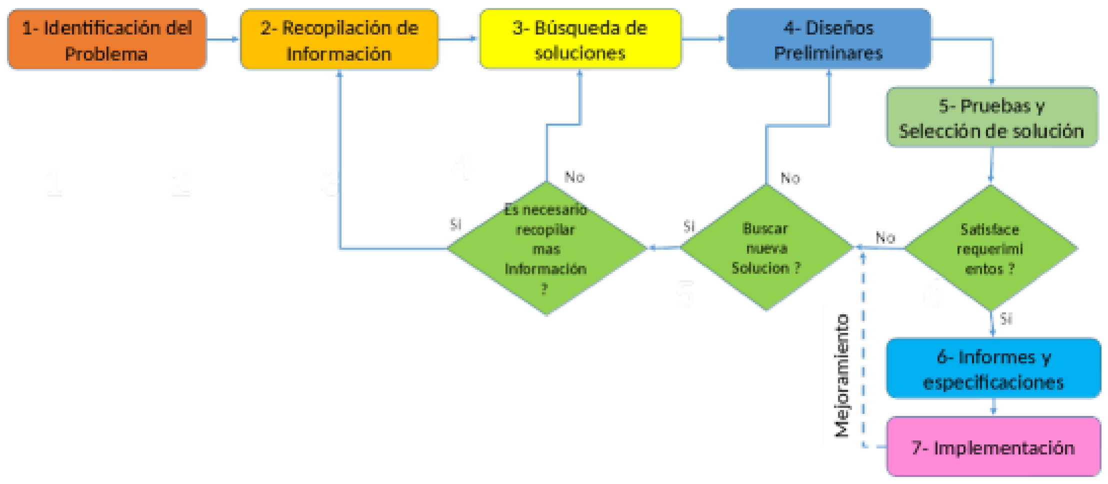
Context of the problem:

The problem in question is the creation of a strategy game based on the conquest of territories. Players must conquer adjacent territories, represented as vertices in a graph, where the edges represent the connections between the territories. The objective is to determine the areas conquered by each player and calculate the corresponding score.

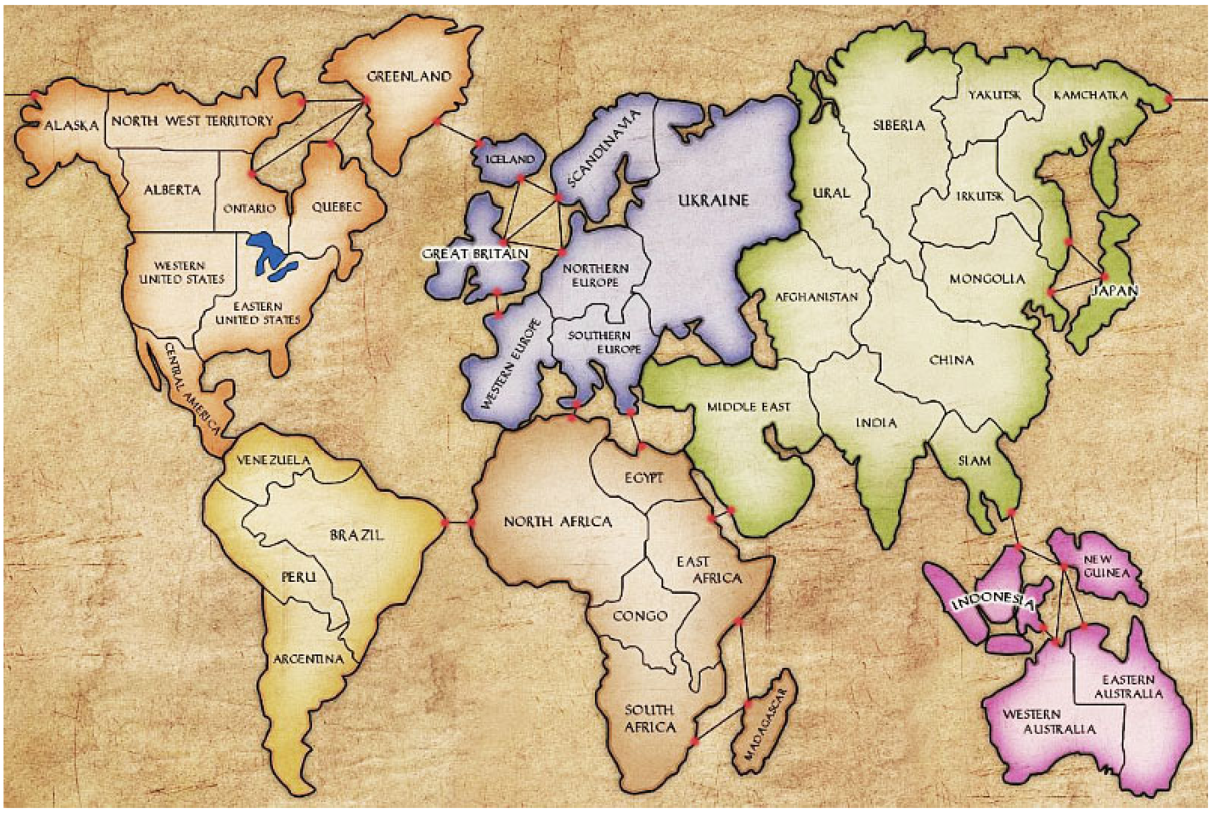
Solution development:

Given the context and nature of the problem posed, we have decided to use the Engineering Method for the development of an effective and efficient solution. This systematic approach will allow us to perform a thorough analysis and understanding of the problem situation, identify the necessary requirements, and set clear and achievable goals for the solution.

Based on the description of the Engineering Method provided in the book "Introduction to Engineering" by Paul Wright, we have defined the following flowchart, which we will follow in the development of the solution:


1- Identificación del Problema
2- Recopilación de Información
3- Búsqueda de soluciones
4- Diseños Preliminares
5- Pruebas y Selección de solución
Satisface requerimientos ?
Si
No
6- Informes y especificaciones
7- Implementación
Mejoramiento
Buscar nueva Solucion ?
Si
No
Es necesario recopilar mas información ?
Si
No

Step 1: Problem Identification:



Problem definition:

The problem of the strategy game based on the conquest of territories centers on the way in which the players can conquer adjacent territories in the context of the graph. In this game, the territories are represented as vertices in the graph, the edges represent the connections between the territories, and the weights of these represent the necessary troops from a territory in that region.

The edges of the graph are essential, since they define the adjacency relations between the territories. Each edge connects two vertices and represents that those territories are adjacent to each other, which means that they can be conquered one after the other in the game. So, the existence of an edge between two vertices indicates the possibility that a player can conquer one of the territories with the troops of that edge and then advance towards the adjacent territory through that connection.

The vertices represent the individual territories in the game. Each vertex of thecount corresponds to a specific territory that players can conquer. The way the vertices are connected by the edges determines which territories are considered adjacent and thus can be conquered by the player.

The problem lies in finding an optimal strategy so that the players conquer adjacent territories in the graph, taking advantage of the connections represented by the edges. Graph traversal algorithms such as BFS and DFS are used to determine the areas conquered by each player and calculate the corresponding score. These algorithms allow players to efficiently explore the graph and make the best decisions.

Symptoms and Needs:

- Lack of an efficient system for managing the process of conquest of territories in the game.
- Absence of a data structure that allows the territories and the connections between them to be adequately represented.
- Lack of algorithms for routes over graphs to determine the areas conquered by each player.

Causes:

- Difficulty for players to strategically conquer adjacent territories.
- Lack of balance in the game, since the scores of each player cannot be calculated correctly.
- Loss of interest and satisfaction on the part of the players when facing a challenging game but with problems in its operation.
- Potential conflicts between players due to lack of clear and fair territory conquest mechanics.
- Limitation in the gaming experience, as players cannot take full advantage of the strategic and competitive features of the game.

Step 2: Information Gathering:

Once the problem has been identified and the needs properly defined, the engineer begins to gather the information and data necessary to solve it. In the case of the current project, the problem revolves around the creation of a strategy game based on the conquest of territories using graphs. The objective is to determine the areas conquered by each player and calculate the corresponding scores.

In this initial phase, comprehensive information on graph-based games, strategies for territory conquest, and graph data structures was collected. Gathering information is crucial to understanding current practices in game development, as well as identifying best practices for designing and implementing conquest mechanisms.

As a first section, various graph data structures suitable to represent the territories and their connections will be explored. Common data structures such as adjacency lists or adjacency matrices will be studied, along with their advantages and disadvantages.

Adjacency Lists: An adjacency list is a data structure that is commonly used to represent graphs. It consists of a list of vertices, where each vertex has associated a list of its adjacent vertices. This representation is efficient in terms of space when the graph is sparse and allows fast access to the adjacent vertices of each vertex.

Bibliographic reference:

Cormen TH, Leiserson CE, Rivest RL, & Stein C (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Adjacency Matrices: An adjacency matrix is a two-dimensional data structure that represents a graph. The matrix has dimensions nxn, where n is the number of vertices in the graph. Each entry in the array represents a connection between two vertices, and a Boolean or numeric value is used to indicate the existence or weight of the edge. This representation is efficient in terms of fast edge access, but may require more space compared to adjacency lists, especially for large or dense graphs.

Bibliographic reference:

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Java (6th ed.). Wiley.

Secondly, it will talk about games on the market similar to the one that you want to develop in order to have a solid foundation for the development of the solution to the problem.

An example of an existing game that bears similarities to the project under development is the popular board game "Risk." In Risk, the game board is made up of a map divided into regions, where each region represents a territory. These regions are connected to each other through shared borders, which establishes a graph structure in the game.

As in the development project, in Risk players must conquer adjacent territories to expand their domain. This is similar to the idea of the vertices of the graph representing the territories in the project. Players in Risk make strategic decisions about which territories to attack and how to deploy their armies, considering the connections between territories and the possibility of advancing into neighboring territories.

Also, in the board game Risk, the concept of conquering entire regions is related to the idea of areas conquered by each player in the project. Players must plan and execute strategies to fully control a region, which results in the conquest of all related territories within the graph.

The inclusion of this example, the board game "Risk", provides a relevant and practical analogy for understanding how adjacent territories and the connections between them can be represented by graphs in the context of the project under development. Likewise, it can serve as a source of inspiration for the implementation of the mechanics of conquest of territories and calculation of conquered areas using the graph structure.

Bibliographic reference:

Hasbro. (1959). Risk [Board game]. Hasbro Gaming.

Likewise, it will be important to investigate graph traversal algorithms to calculate the areas conquered by each player. Depth Search (DFS) and Breadth Search (BFS) are examples of algorithms that can be used to efficiently explore connected territories.

Depth Search (DFS): Depth search algorithm is a technique used to explore or traverse a graph. Start at an initial node, and from there, explore as deeply as possible along each branch before backtracking. It uses a stack to keep track of the nodes that need to be visited. DFS is useful for finding paths, detecting cycles, and performing searches in unweighted graphs.

Bibliographic reference:

Cormen TH, Leiserson CE, Rivest RL, & Stein C (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Breadth-Finding Search (BFS): The breadth-finishing algorithm is another technique used to traverse or explore a graph. It starts at an initial node and explores all of that node's neighbors before moving on to neighbors of neighbors. It uses a GraphAdjacencyList or GraphMatrix to keep track of the nodes that need to be visited. BFS is useful for finding the shortest path between two nodes, discovering connected components, and performing searches in unweighted graphs.

Bibliographic reference:

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Java (6th ed.). Wiley.

In conclusion, by collecting information from various sources such as research articles, books, and online resources, you will establish a strong knowledge base. This knowledge will be used to propose adequate and efficient solutions to the problem at hand, including the representation of territories as vertices and the use of graph algorithms to calculate conquered areas and scores.

Step 3: Creative solutions search: In this phase, the creation of creative solution ideas for the problem is intended:

It is important to generate a preliminary list of solutions to have a more accurate approximation to the problem. This phase is fundamental to solve the problem effectively, and it is also important for steps 4 and 5 of the engineering method, where less feasible ideas are discarded, and the best solution is selected. The technique used for the project was brainstorming, which is based on the free and spontaneous generation of ideas without judging them, and in the subsequent selection and combination of those considered most relevant and useful. The team members were asked to propose solution ideas based on their research and knowledge of the problem, and the resulting solution ideas are as follows:

- Random Starting Positions: Implement a mechanism where the starting positions of the territories assigned to the players are random in each game. This adds unpredictability and strategic diversity to each game, making it more challenging and entertaining.

- Special Abilities: Introduce special abilities or upgrades that players can acquire during the game. These abilities can provide unique advantages such as faster troop movement, increased defense, or the ability to conquer multiple territories in a single turn. This adds a layer of complexity and strategic decision making to the game.

- Diplomatic Negotiations: Allow players to negotiate alliances, treaties, and trade agreements with each other. This introduces a diplomatic aspect to the game, where players can form temporary alliances to conquer territories together or negotiate a peaceful coexistence to focus on other strategic objectives.

- Territory Improvements: Allow players to improve conquered territories by investing resources. These upgrades could provide defensive bonuses, increases in resource production, or additional troop recruiting capabilities. This incentivizes players to focus on specific territories and strategically manage their resources.

- Event Cards: Introduce a deck of event cards that can be drawn during the game. These cards can have various effects, such as changing the borders between territories, triggering natural disasters, or providing temporary buffs or setbacks for players. This adds unpredictability and strategic adaptability to the game.

- Multiple Victory Conditions: Instead of a single victory condition (for example, conquer all territories), introduce multiple ways to win the game. For example, players can achieve victory by conquering a certain number of regions, forming alliances with other players, or completing specific objectives. This diversifies the game and allows for different strategic approaches.

It must be taken into account that the previous list of solutions was proposed based on the characteristics that the solution to the problem could have, because the game structure is already defined (it will not be changed in the development of the solution), therefore, in the solution proposals an attempt was made to propose a feature that would differentiate the game and give it a degree of originality, which is one of the main objectives of the project.

Step 4: Transition from idea formulation to preliminary designs:

In this phase, ideas that are not feasible will be discarded, and promising ideas will be shaped and modified to form feasible drafts and designs.

First, we will start with the ideas that will be discarded, which are as follows:

- Diplomatic Negotiations: If the main objective of the game is to focus on military strategy and the conquest of territories, diplomatic negotiations could divert attention from that central focus. If you're looking for a game more oriented towards direct competition and tactical decision-making on the battlefield, discarding diplomatic negotiations can keep the game focused on those aspects.

- Event Cards: If you want to keep the game more predictable and strategic, event cards can introduce too much uncertainty and unpredictability into the game. If you want players to plan and execute long-term strategies, event cards could generate random outcomes that make it difficult to make strategic decisions and long-term planning. Dismissing this idea will help keep the game more focused on strategic planning and execution.

- Territory Improvements: If the game is primarily focused on territory conquest and direct interaction between players, territory improvements can add an additional layer of complexity and resource management that can divert attention from the main objective. If you're looking to keep the game more streamlined and oriented towards direct military strategy, discarding the improvements in territories can simplify the game mechanics and focus it on the conquest of territories itself.

In conclusion, when analyzing the different solution ideas proposed for the problem of the conquest of territories in the strategy game based on graphs, three of them have been discarded: diplomatic negotiations, event cards and improvements in the territories. These decisions are based on the need to maintain the strategic focus of the game, either through direct competition in the conquest of territories or long-term strategic planning. By discarding these ideas, it seeks to maintain a balance in the game, avoid possible imbalances between players and guarantee an experience focused on making tactical decisions and executing military strategies. With this selection of solutions, it will be possible to advance in the next stage of the engineering process to choose the best solution and continue the development of the game.

Next, we will discuss the ideas that are not discarded, specifying them and making their respective analytical specification models:

Random starting positions:

This idea proposes to assign the starting positions of the players randomly on the game map. At the start of the game, each player would be given a random territory to begin their conquest. The advantage of this mechanic is that it introduces an element of surprise and variety to each game, requiring players to adapt their strategies based on the initial circumstances. However, a potential downside is that randomization can lead to initial imbalances, as some players might start out in more favorable territory than others. This can affect the competitiveness and equality of opportunity between the players.

Analytical model:

1. Definition of variables:

- N: Total number of players
- Map: Representation of the game map
- Players: List of players
- Territories: List of available territories
- Assignments: Random assignments of starting territories for players

2. Generation of random assignments:
   - Create a list of available territories.
   - For each player in the player list:
     - Randomly choose a territory from the list of available territories.
     - Assign that territory to the player.
     - Remove that territory from the list of available territories.
3. Verification of initial imbalances:
   - Calculate a balance metric (for example, add the characteristics of the territories assigned to each player).
   - Calculate the mean and standard deviation of the equilibrium metric for the initial assignments.
   - If the standard deviation exceeds a predefined threshold, it is considered that there are initial imbalances.
4. Management of imbalances:

- If there are initial imbalances:
    - Make adjustments to territory assignments to balance the game (for example, swap territories between players).
    - Provide bonuses to disadvantaged players to offset the initial imbalance.
- If there are no initial imbalances, continue with the initial assignments.

5. Start of the game:
   - Show the game map with the starting positions of the players.
   - Start the game.
6. Development of the game:
   - Players carry out their actions according to the rules of the game.
   - Continue the game until an end condition is met (for example, full conquest, time up, etc.).
7. Game over:
   - Determine the final result of the game.
   - Show the results (points, classification, etc.).
   - Finish the game.

This analytical model provides a general structure to implement the proposed idea. Generating random assignments, checking for initial imbalances, and handling imbalances are important steps in ensuring competitiveness and equal opportunity among players. The development and the end of the gameit will depend of the specific rules and mechanics of the game in question.

Special abilities:

The idea of special abilities is to assign each player unique or special abilities that give them strategic advantages during the game. Each player would have a different ability that could be used at strategic moments to influence the development of the game. These abilities can vary, from increasing the defense of one's own territories, obtaining additional resources, sabotaging the opponent's movements, among other possibilities. Special abilities add diversity and customization to the game, allowing players to explore different strategies and tactics based on their unique abilities. However, it is important to balance these skills to avoid any of them being too powerful or unbalanced compared to the others, which could affect gameplay and competitiveness.

Analytical Model:

1. Definition of variables:
   - N: Total number of players
   - Players: List of players
   - Skills: List of special skills available
   - Assignments: Special skill assignments for players
2. Generation of special ability assignments:
   - Create a list of special abilities available.
   - For each player in the player list:
     - Randomly choose a special skill from the list of available skills.
     - Assign that special ability to the player.
     - Remove that special skill from the list of available skills.
3. Balance of special abilities:
   - Evaluate each special ability and its impact on the game.

- Assign a level of power or advantage to each special ability.
- Verify that there is no significant disproportion in power levels between the special abilities assigned to the players.
- If an imbalance is detected, make adjustments to power levels or assigned abilities to balance the game.

4. Start of the game:
   - Show the special abilities assigned to each player.
   - Start the game.

5. Development of the game:
   - Players use their special abilities at strategic moments during the game.
   - Each player must abide by the rules and restrictions associated with their special ability.
   - Special abilities can influence the development of the game, such as increasing defense, obtaining additional resources or sabotaging the opponent's movements.
6. Balance during the game:
   - Monitor the impact of special abilities during the game.
   - If any special ability becomes too powerful or unbalanced, consider adjustments to keep the game competitive and fun.
   - Make changes to special abilities, restrictions or associated rules if necessary.
7. Game over:
   - Determine the final result of the game, considering strategic actions and the use of special abilities.
   - Show the results and reward the players according to their performance.
   - Finish the game.

This analytical model provides a general framework for implementing the idea of special abilities. Generating random assignments, balancing skills, and in-game monitoring are important steps in ensuring a balanced and fun gameplay experience. Each special ability can provide unique strategic advantages, allowing players to explore different tactical approaches. However, it is essential to balance the special abilities to avoid any of them being too powerful and unbalanced, which could negatively affect the gameplay experience and the competitiveness between players.

<u>Multiple victory conditions:</u>

Instead of having a single win condition, this idea proposes setting different conditions that players must meet to win the game. For example, apart from the conquest of territories, conditions such as accumulating a certain amount of resources, completing specific objectives or achieving an alliance with other players could be included. The implementation of multiple victory conditions adds strategic depth to the game, as players must consider different approaches and tactics to reach their goal. However, it is important to ensure that the conditions are balanced and avoiding to help too much of a type of strategy or a particular player, to maintain competition and challenge among the participants.

<u>Analytical Model:</u>

1. Definition of variables:
   - N: Total number of players
   - Players: List of players
   - Conditions: List of available victory conditions
   - Objectives: Specific objectives associated with victory conditions
   - Progress: Each player's progress towards victory conditions

2. Establishing Victory Conditions:
   - Create a list of available victory conditions.
   - For each victory condition, define the associated specific objectives.
   - Assign a level of difficulty or requirements for each objective.

3. Start of the game:
   - Assign victory conditions to players.
   - Show the conditions and objectives assigned to each player.

- Start the game.

4. Development of the game:
   - Players carry out their actions and strategies to meet the objectives and the assigned victory conditions.
   - Objectives can include the conquest of territories, accumulation of resources, completing specific missions or establishing alliances with other players.

5. Tracking progress towards victory conditions:
   - Record the progress of each player towards the assigned objectives.
   - Update the progress based on the actions performed by the players during the game.

6. Balancing Victory Conditions:
   - Evaluate the conditions and objectives assigned to each player.
   - Verify that there is no significant disproportion in difficulty or requirements between the victory conditions.
   - Make adjustments to the requirements or assigned conditions if an imbalance is detected to maintain competition and challenge among players.

7. Verification of victory conditions:
   - Check if any player has fulfilled all the assigned victory conditions.
   - If a player fulfills all his victory conditions, he is declared the winner of the game.

8. Game over:
   - Show the results and reward the winning player.
   - Finish the game.

This analytical model provides a general framework for implementing the idea of multiple victory conditions. The conditions and objectives assigned to each player allow for greater strategic depth, as players must consider different approaches and tactics to meet their specific objectives. Balancing victory conditions is essential to avoid overly favoring a particular strategy type or player, thus maintaining competition and challenge among participants.

Step 5: Evaluation and Selection of the Best Solution:

Finally, in this writing, the selection of the best solution will be carried out. For this purpose, a series of criteria were established that will be scored from 1 to 5 (where 5 is the highest score considered excellent and 1 is the lowest score considered regular) and will allow choosing the best solution proposal. The criteria are as follows:

- Efficiency: Which of the proposed solutions is more efficient in terms of time and resources?

- Ease of implementation: Which of the solutions is easier to implement and maintain? Is research necessary to solve the problem?

- Scalability: Capacity of the strategy game system based on the conquest of territories to handle an increase in the number of players, territories and connections.

- Usability: Is the solution easy to use for end-users? Is it easy to understand and use for users?

- Security: Does the solution guarantee the privacy and security of the users?

- Cost: What is the cost of implementing and maintaining the solution? Is it financially viable?

The methodology used to evaluate the criteria based on the solution proposals was to have the team meet and agree on a score (1-5) for each criterion. The results were as follows:

Random starting positions:

Criteria	***Score (1-5)***
Efficiency	5
Ease of implementation	4
Scalability	5
Usability	5
Security	5
Cost	5
Total	29

<u>Special abilities:</u>

Criteria	***Score (1-5)***
Efficiency	3
Ease of Implementation	3
Scalability	4
Usability	5
Security	4
Cost	4
Total	23

Multiple victory conditions:

Criteria	***Score (1-5)***
Efficiency	3
Ease of Implementation	4
Scalability	5
Usability	5
Security	4
Cost	4
Total	24

Based on the scores obtained, the winning criteria obtained a total of 29, the highest score among the ideas evaluated. This indicates that it best meets the established criteria and is considered the most suitable option for the project in terms of efficiency, ease of implementation, scalability, usability, security and cost.

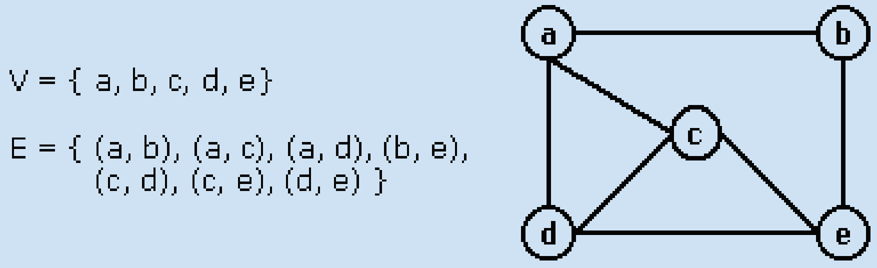
In conclusion, after evaluating different ideas based on several criteria, the idea of random starting positions has been chosen as the optimal solution for the strategy game based on the conquest of territories. This idea scored highest for efficiency, scalability, usability, security, and cost, making it the most appropriate choice for the project.

In this solution, the game will be implemented using the Java programming language and making use of graph-based data structures. The idea of random starting positions involves randomly assigning starting territories to each player at the start of the game. This adds an element of surprise and strategy, as players don't know which territories are theirs and must adapt their strategy based on their assigned locations. The use of graphs will allow representing the connections between territories, which will facilitate the calculation of conquered areas and the determination of victory conditions. Furthermore, development in Java will provide flexibility and ease of implementation.

In summary, the idea of random starting positions has been selected due to its high score on the evaluated criteria and its ability to provide an exciting and challenging experience. With the use of graphs and the implementation in Java, it is expected to create a strategy game based on the conquest of territories that is efficient, scalable and easy to play.

TAD DEFINITION

TAD <Graph>


V = { a, b, c, d, e }

E = { (a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e) }

Graph =

Invariant:

Let G=(V, E) be a graph, where V is the set of vertices and E is the set of edges.

- For every vertex v in V, there exists a unique identifier ID(v) that represents it.
- For every pair of vertices u and v in V, there exists a path from u to v through a sequence of edges in E.
- For every edge e in E, the start and end vertices of e are different, that is, e = (u, v) where u ≠ v.
- For every edge e in E, the start and end vertices of and belong to V, that is, e = (u, v) where u, v ∈ V.
- For every edge e in E, there is a weight w(e) associated to It is, y w(e) ≥ 0.
- For every edge e = (u, v) in E, there exists an edge e' = (v, u) in E, which implies an arrival connection from u to v and an arrival connection from v to u.

Main operations:

CreateGraph(): N/A → Graph

AddVertex(): Graph x Element → Graph

RemoveVertex(): Graph x Element → Element

AddEdge(): Graph x ElementOrigin x ElementFinish x Weight → Graph

RemoveEdge(): Graph x ElementOrigin x ElementFinish → Edge

GetAdjacent(): Graph x Element → Elements

GetWeightEdge(): Graph x ElementOrigin x ElementFinish → Integer

GetVertexs(): Graph → Elements

IsVertex(): Graph x Element → boolean

IsEdge(): Graph x ElementOrigin x ElementFinish → boolean

Specifications:

CreateGraph(): Construction Operation

“Create a new empty Graph”

{pre:True}

{pos: New Graph}}

AddVertex(): Modifying Operation

“Add a vertex to a Graph”

{pre: Graph ≠ nil }

{pos: Elemento ∈ Graph}}

RemoveVertex(): Modifying Operation

“Deletes a vertex from a Graph”

{pre: Graph ≠ nil ^ Elemento ∈ Graph }

{pos: Elemento ∉ Graph}}

AddEdge(): Modifying Operation

“Add an Edge between one registered vertex and another”

{pre: Graph ≠ nil ^Element Origin ∈ Graph ^ ElementoFinish ∈ Graph ^ Weight >= 0}

{pos: {ElementOrigin, ElementFinish} ∈ Graph}

RemoveEdge(): Modifying Operation

“Deletes an Edge, but not the vertices that compose it”

{pre: Graph ≠ nil ^Element Origin ∈ Graph ^ ElementoFinish ∈ Graph ^ {ElementOrigin, ElementFinish} ∈ Graph}} {pos: {ElementOrigin, ElementFinish} ∉ Graph ^ ElementoOrigin ∈ Graph ^ ElementoFinish ∈ Graph}

GetAdjacent(): Parser Operation

“Get the list of Vertexs adjacent of a given Vertex”

{pre: Graph ≠ nil ^ Element ∈ Graph}

{pos: Elements}

GetWeightEdge(): Parser Operation

“Get the weight of a Edge registered”

{pre: Graph ≠ nil ^Element Origin ∈ Graph ^ ElementFinish ∈ Graph ^ {ElementOrigin, ElementFinish} ∈ Graph}

{pos: weight}

GetVertexs(): Parser Operation

“Get list of vertexs of Graph”

{pre: Graph ≠ nil}

{pos: Elements}

IsVertex(): Parser Operation

“Check if the vertex exists”

{pre: Graph ≠ nil}

{pos: True if Element ∈ Graph, False otherwise}

IsEdge(): Parser Operation

“Check if the Edge exists”

{pre: Graph ≠ nil}

{pos: True yesElementOrigin ∈ Graph ^ ElementFinish ∈ Graph ^ {ElementOrigin, ElementFinish} ∈ Graph, False otherwise}

TESTS

Graph Adjacency List and Graph Matrix Tests

The tests presented below are applicable to both the adjacency list graph and the adjacency matrix graph. The implementation used corresponds to each of them, respectively. By running the tests on both types of graphs, the expected results are verified, ensuring the proper functioning of the implementations.

Name	Class	Scenery
setUpNotDirected	Graph Adjacency ListTest	The Graph is empty and it is a graph not directed.

Name	Class	Scenery
setUpDirected	Graph Adjacency ListTest	The Graph is empty and it is a graph directed.

Test Objective: Verify the function Dijkstra in the GraphAdjacencyList and GraphMatrix.

Class	Method	Scenery	Entry Values	Expected result
GraphAdjacencyList and GraphMatrix	testDijkstra1	setupNotDirected()	graph.addVertex(1)	The graph has a vertex (1) with a distance of 0.0
GraphAdjacencyList and GraphMatrix	testDijkstra2	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addVertex(5), graph.addEdge(1, 2, 3), graph.addEdge(1, 3, 5), graph.addEdge(2, 3,	The graph have some vertexs and edges with their distances

			2), graph.addEdge(2, 4, 1), graph.addEdge(3, 4, 2), graph.addEdge(3, 5, 4), graph.addEdge(4, 5, 3)	
GraphAdjacencyList and GraphMatrix	testDijkstra3	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3)	The graph has several vertices with infinite distances from vertex 1

Test Objective: Verify the function Kruskal in the GraphAdjacencyList and GraphMatrix.

Class	**Method**	**Scenery**	**Entry Values**	**Expected result**
GraphAdjacencyList and GraphMatrix	testKruskal1	setupNotDirected()	graph.addVertex(1)	The minimum spanning tree is empty
GraphAdjacencyList and GraphMatrix	testKruskal2	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3)	The minimum spanning tree is empty
GraphAdjacencyList and GraphMatrix	testKruskal3	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addVertex(5), graph.addEdge(1, 2, 1), graph.addEdge(1, 3, 3), graph.addEdge(2, 3, 4), graph.addEdge(2, 4, 6), graph.addEdge(3, 4, 5), graph.addEdge(3,	The minimum spanning tree has 4 edges with the expected weights and vertices

			5, 2), graph.addEdge(4, 5, 7)	

Test Objective: Verify the function Prim in the GraphAdjacencyList and GraphMatrix.				
Class	**Method**	**Scenery**	**Entry Values**	**Expected result**
GraphAdjacencyList and GraphMatrix	cf.1	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addEdge(1, 2, 1), graph.addEdge(1, 3, 2), graph.addEdge(2, 3, 3), graph.addEdge(2, 4, 4), graph.addEdge(3, 4, 5)	The minimum spanning tree has 3 edges.
GraphAdjacencyList and GraphMatrix	prim2	setupNotDirected()	graph.addVertex(1), graph.addVertex(2)	The result is null
GraphAdjacencyList and GraphMatrix	prim3	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addVertex(5), graph.addEdge(1, 2, 1), graph.addEdge(1, 3, 2), graph.addEdge(2, 3, 3), graph.addEdge(2, 4, 4), graph.addEdge(3, 4, 5), graph.addEdge(4, 5, 10)	The tree has 4 edges

Test Objective: Verify the function PrintPrim in the GraphAdjacencyList and GraphMatrix.				
Class	**Method**	**Scenery**	**Entry Values**	**Expected result**
GraphAdjacencyList and GraphMatrix	testPrintPrim1	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 5), graph.addEdge(2, 3, 3), graph.addEdge(1, 3, 4)	The minimum spanning tree contains 2 edges with the expected weights and vertices and the output is the expected string
GraphAdjacencyList and GraphMatrix	testPrintPrim2	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addVertex(5), graph.addEdge(1, 2, 5), graph.addEdge(1, 3, 2), graph.addEdge(2, 3, 4), graph.addEdge(2, 4, 1), graph.addEdge(3, 4, 6), graph.addEdge(3, 5, 3), graph.addEdge(4, 5, 2)	The minimum spanning tree contains 5 edges with the expected weights and vertices and the output is the expected string.
GraphAdjacencyList and GraphMatrix	testPrintPrim3	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addEdge(1, 2, 5)	Vertex 3 does not exist in the graph and the output is the expected string

Test Objective: Verify the function AddEdge in the GraphAdjacencyList and GraphMatrix.

Class	Method	Scenery	Entry Values	Expected result
GraphAdjacencyList and GraphMatrix	testAddEdge 1	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addEdge(1, 2, 30)	The graph has 1 edge in each of the vertices, the vertices of the edges have the expected values
GraphAdjacencyList and GraphMatrix	testAddEdge 2	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addEdge(1, 2, 30)	The first vertex has 1 outgoing edge, the second vertex has no edges, and the values are as expected.
GraphAdjacencyList and GraphMatrix	testAddEdge 3	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addEdge(1, 3, 30)	None of the vertices have edges and the connection between vertices 1 and 3 is not made

Test Objective: Verify the function AddVertex in the GraphAdjacencyList and GraphMatrix.

Class	Method	Scenery	Entry Values	Expected result
GraphAdjacencyList and GraphMatrix	testAddVertex1	setupNotDirected()	graph.addVertex(1)	The graph has 1 vertex, whose value is 1
GraphAdjacencyList and GraphMa	testAddVertex2	setupDirected()	graph.addVertex(1), graph.addVertex(1), graph.addVertex(3)	The graph has 2 vertices with the values 1 and 3 respectively.

trix				
GraphAdjacencyList and GraphMatrix	testAddVertex3	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4)	The graph has 4 vertices with the values 1, 2, 3 and 4 respectively.

Test Objective: Verify the function GetAdjacentAsString in the GraphAdjacencyList and GraphMatrix.

Class	Method	Scenery	Entry Values	Expected result
GraphAdjacencyList and GraphMatrix	testGetAdjacentVerticesAsString1	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addEdge(1, 2, 0)	Vertex 3 does not exist in the graph
GraphAdjacencyList and GraphMatrix	testGetAdjacentVerticesAsString2	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 0), graph.addEdge(1, 3, 0)	Vertex 1 has adjacent 2 and 3
GraphAdjacencyList and GraphMatrix	testGetAdjacentVerticesAsString3	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3)	Vertex 2 has no adjacent

Test Objective: Verify the function RemoveEdge in the GraphAdjacencyList and GraphMatrix.

Class	Method	Scenery	Entry Values	Expected result
GraphAdj	testRemove	setupDirect	graph.addVertex(1),	After deleting the edge between vertex

acencyList and GraphMatrix	Edge1	ed()	graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 30), graph.addEdge(2, 3, 30)	2 and 3, vertex 2 must have no adjacent
GraphAdjacencyList and GraphMatrix	testRemoveEdge2	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 30), graph.addEdge(2, 3, 30)	After deleting the edge between vertex 1 and 2, vertex 1 should have no adjacent, and vertex 2 should have an adjacent to vertex 3
GraphAdjacencyList and GraphMatrix	testRemoveEdge3	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 30)	After trying to delete the edge between vertex 1 and 3 (which does not exist), there should be no change in the graph

Test Objective: Verify the function RemoveVertex in the GraphAdjacencyList and GraphMatrix.

Class	**Method**	**Scenery**	**Entry Values**	**Expected result**
GraphAdjacencyList and GraphMatrix	testRemoveVertex1	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 30), graph.addEdge(2, 3, 30)	After removing vertex 2, the graph must have 2 vertices and none of them must have adjacent ones
GraphAdjacencyList and GraphMatrix	testRemoveVertex2	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 30)	After trying to remove vertex 4 (which does not exist), the graph should not be changed
GraphAdjacencyList and GraphMatrix	testRemoveVertex3	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 30), graph.addEdge(2, 3,	After removing vertex 2, the graph must have 2 vertices and none of them must have adjacent ones

			30)	

Test Objective: Verify the function FloydWarshall in the GraphAdjacencyList and GraphMatrix.				
Class	**Method**	**Scenery**	**Entry Values**	**Expected result**
GraphAdjacencyList and GraphMatrix	testFloydWarshall1	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 1), graph.addEdge(2, 3, 2), graph.addEdge(1, 3, 3)	A distance matrix is expected that reflects the shortest distances between the vertices of the graph.
GraphAdjacencyList and GraphMatrix	testFloydWarshall2	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addEdge(1, 2, 1), graph.addEdge(2, 3, 2), graph.addEdge(3, 4, 1), graph.addEdge(1, 4, 10)	A distance matrix is expected that reflects the shortest distances between the vertices of the graph
GraphAdjacencyList and GraphMatrix	testFloydWarshall3	setupDirected()	No vertices or edges are added to the graph	An empty distance matrix is expected, since there are no vertices or edges in the graph.

Test Objective: Verify the function DFS in the GraphAdjacencyList and GraphMatrix.				
Class	**Method**	**Scenery**	**Entry Values**	**Expected result**
GraphAdjacencyList and GraphMatrix	testDFS1	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addVertex(5), graph.addEdge(1, 2, 30), graph.addEdge(1, 3,	All the vertices of the graph are expected to be black after applying the DFS algorithm.

			30), graph.addEdge(2, 4, 30), graph.addEdge(3, 4, 30), graph.addEdge(4, 5, 30)	
GraphAdjacencyList and GraphMatrix	testDFS2	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addVertex(5), graph.addEdge(1, 2, 30), graph.addEdge(1, 3, 30), graph.addEdge(2, 4, 30), graph.addEdge(3, 4, 30), graph.addEdge(4, 5, 30)	The vertex completion time and distance values are expected to be correct after applying the DFS algorithm on a directed graph
GraphAdjacencyList and GraphMatrix	testDFS3	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addVertex(5), graph.addEdge(1, 2, 30), graph.addEdge(1, 3, 30), graph.addEdge(2, 4, 30), graph.addEdge(3, 4, 30), graph.addEdge(4, 5, 30)	The predecessor values of the vertices are expected to be correct after applying the DFS algorithm on an undirected graph. The first vertex is expected to have no predecessor.

Test Objective: Verify the function SearchVertex in the GraphAdjacencyList and GraphMatrix.

Class	**Method**	**Scenery**	**Entry Values**	**Expected result**

GraphAdjacencyList and GraphMatrix	testSearchVertex1	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addEdge(1, 2, 0), graph.addEdge(2, 3, 0), graph.addEdge(3, 4, 0)	The index returned for vertex 3 is expected to be 2, since it is at position 2 of the vertex array.
GraphAdjacencyList and GraphMatrix	testSearchVertex2	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addEdge(1, 2, 0), graph.addEdge(2, 3, 0), graph.addEdge(3, 4, 0)	The return index for vertex 5 is expected to be -1, since it is not found in the vertex array
GraphAdjacencyList and GraphMatrix	testSearchVertex3	setupNotDirected()	N/A	The return index for vertex 1 is expected to be -1, since no vertex has been added to the graph.

Test Objective: Verify the function BFS in the GraphAdjacencyList and GraphMatrix.				
Class	**Method**	**Scenery**	**Entry Values**	**Expected result**
GraphAdjacencyList and GraphMatrix	testBFS1	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addVertex(5), graph.addEdge(1, 2, 30), graph.addEdge(1, 3, 30), graph.addEdge(2, 4, 30), graph.addEdge(3, 4, 30),	The calculated distances for each vertex from vertex 1 are expected to be: 0, 1, 1, 2, 3.

			graph.addEdge(4, 5, 30)	
GraphAdjacencyList and GraphMatrix	testBFS2	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addVertex(5), graph.addEdge(1, 2, 30), graph.addEdge(1, 3, 30), graph.addEdge(2, 4, 30), graph.addEdge(3, 4, 30), graph.addEdge(4, 5, 30)	All vertices visited during the BFS traversal are expected to have the color BLACK.
GraphAdjacencyList and GraphMatrix	testBFS3	setupNotDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addVertex(4), graph.addEdge(1, 2, 30), graph.addEdge(2, 3, 30)	The computed distances for each vertex from vertex 1 are expected to be: 0, 1, 2 and vertex 4 is expected to have infinite distance (represented by Integer.MAX_VALUE).

Test Objective: Verify the function PrintGraph in the GraphAdjacencyList and GraphMatrix.				
Class	**Method**	**Scenery**	**Entry Values**	**Expected result**
GraphAdjacencyList and GraphMatrix	testPrintGraph1	setupDirected()	N/A	The graph with all its components
GraphAdjacencyList and GraphMatrix	testPrintGraph2	setupDirected()	graph.addVertex(1)	The graph with all its components
GraphAdjacencyList and GraphMatrix	testPrintGraph3	setupDirected()	graph.addVertex(1), graph.addVertex(2), graph.addVertex(3), graph.addEdge(1, 2, 5), graph.addEdge(2, 3, 3)	The graph with all its components

Test Objective: Verify the function searchEdge in the GraphAdjacencyList and GraphMatrix.

Class	Method	Scenery	Entry Values	Expected result
GraphAdjacencyList and GraphMatrix	testSearchEdge1	setupNotDirected()	graph.addVertex(1); graph.addVertex(2); graph.addEdge(1, 2, 30); Edge<Integer> edge = graph.searchEdge(1, 2);	Find the edge created through the find method
GraphAdjacencyList and GraphMatrix	testSearchEdge2	setupDirected()	graph.addVertex(1); Edge<Integer> edge = graph.searchEdge(1, 2);	Find the edge created through the find method
GraphAdjacencyList and GraphMatrix	testSearchEdge3	setupNotDirected()	graph.addVertex(1); graph.addVertex(2); graph.addEdge(1, 2, 30); Edge<Integer> edge = graph.searchEdge(1, 2);	Find the edge created through the find method

Tests of the game

We are pleased to present the testing phase for the Territory Conquest Game. In this phase, we will thoroughly evaluate the functionality and performance of the game to ensure a flawless and engaging gaming experience for our users. The tests will cover a wide range of aspects, including gameplay mechanics, user interactions, and the accuracy of game features.

Name	Class	Scenery
setUp	GameTest	The Game start with its countries and edges

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify addition of countries to GraphAdjacencyList and GraphMatrix	setUp()	managerPersistence.getCountries().split("\n")	["USA", "Canada", "Mexico"]	The countries "USA", "Canada", and "Mexico" are added as vertices to graphAdjacencyList and graphMatrix
Verify addition of countries to GraphAdjacencyList and GraphMatrix	setUp()	managerPersistence.getCountries().split("\n")	["Japan", "China", "South Korea"]	The countries "Japan", "China", and "South Korea" are added as vertices to

				graphAdjacencyList and graphMatrix
Verify addition of countries to GraphAdjacencyList and GraphMatrix	setUp()	managerPersistence.getCountries().split("\n")	["Germany", "France", "Italy"]	The countries "Germany", "France", and "Italy" are added as vertices to graphAdjacencyList and graphMatrix

Test Objective	**Scenery**	**Configuration**	**Inputs**	**Expected Result**
Verify retrieval of player name at a valid index	setUp()	players = [Player1, Player2, Player3]	player = 1	Returns the name of Player2
Verify retrieval of player name at index 0	setUp()	players = [Player1, Player2, Player3]	player = 0	Returns the name of Player1

Verify retrieval of player name at the last index	setUp()	players = [Player1, Player2, Player3]	player = 2	Returns the name of Player3
Verify retrieval of player name with an empty players array	setUp()	players = []	player = 0	Throws an exception or returns null (depending on implementation)

Test Objective	**Scenery**	**Configuration**	**Inputs**	**Expected Result**
Verify game finish when a player has fewer troops than the minor troop count	setUp()	players = [Player1, Player2]	players[0].setTroops(5), players[1].setTroops(10), edgeWithMinorTroops() returns 8	Returns true

Verify game finish when a player has conquered 20 or more countries	setUp()	players = [Player1, Player2]	players[0].setConquistedCountries([Country1, Country2, ..., Country20]), players[1].setConquistedCountries([Country21, Country22, ..., Country30])	Returns true
Verify game not finished when neither condition is met	setUp()	players = [Player1, Player2]	players[0].setTroops(15), players[1].setTroops(12), players[0].setConquistedCountries([Country1, Country2, ..., Country19]), players[1].setConquistedCountries([Country20, Country21, ..., Country29])	Returns false

Test Objective	**Scenery**	**Configuration**	**Inputs**	**Expected Result**

Verify game finish message when Player 1 has fewer troops than the minor troop count	setUp()	players = [Player1, Player2]	players[0].setTroops(5), players[1].setTroops(10), edgeWithMinorTroops() returns 8	Returns "The player Player1 hasn't troops."
Verify game finish message when Player 2 has fewer troops than the minor troop count	setUp()	players = [Player1, Player2]	players[0].setTroops(15), players[1].setTroops(4), edgeWithMinorTroops() returns 6	Returns "The player Player2 hasn't troops."

Verify game finish message when Player 1 has conquered 20 or more countries	setUp()	players = [Player1, Player2]	players[0].setConquistedCountries([Country 1, Country2, ..., Country20]), players[1].setConquistedCountries([Country 21, Country22, ..., Country30])	Returns "The player Player1 won by conquering more territories"
Verify game finish message when Player 2 has conquered 20 or more countries	setUp()	players = [Player1, Player2]	players[0].setConquistedCountries([Country 1, Country2, ..., Country19]), players[1].setConquistedCountries([Country 20, Country21, ..., Country30])	Returns "The player Player2 won by conquering more territories"
Verify default game finish message when none of the conditions are met	setUp()	players = [Player1, Player2]	players[0].setTroops(15), players[1].setTroops(12), players[0].setConquistedCountries([Country 1, Country2, ..., Country19]), players[1].setConquistedCountries([Country 20, Country21, ..., Country29])	Returns Skadoosh

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify retrieval of the smallest edge weight in an empty graph	setUp()	graphAdjacencyList = empty graph	N/A	Returns Double.POSITIVE_INFINITY
Verify retrieval of the smallest edge weight in a graph with edges	setUp()	graphAdjacencyList = graph with edges	N/A	Returns the weight of the smallest edge in the graph
Verify retrieval of the smallest edge weight in a graph with equal edge weights	setUp()	graphAdjacencyList = graph with equal edge weights	N/A	Returns the weight of any edge in the graph (since they are all equal)

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify conquering a territory in the adjacency list graph	setUp()	graphAdjacencyList, players	player = 0, newTerritoryName = "Territory2"	Returns the success message with updated player information
Verify conquering a territory in the matrix graph	setUp()	graphMatrix, players	player = 1, newTerritoryName = "Territory1"	Returns the success message with updated player information
Verify conquering a non-existing territory in the adjacency list graph	setUp()	graphAdjacencyList, players	player = 0, newTerritoryName = "NonExisting"	Returns "The territory doesn't exist or is misspelled"

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify printing of the militar strategy for a valid minimum spanning tree	setUp()	minimunSpanningTree	minimunSpanningTree = [Edge1, Edge2, Edge3]	Returns a string representation of the minimum spanning tree with territories and troops
Verify handling of a null minimum spanning tree input	setUp()	N/A	minimunSpanningTree = null	Returns "Mistake with military strategy"
Verify printing of the militar strategy for an empty minimum spanning tree	setUp()	minimunSpanningTree = empty list	minimunSpanningTree = []	Returns an empty string

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify retrieval of the number of troops for player 0	setUp()	players	numPlayers = 0	Returns "Troops of the player: [number of troops]"
Verify retrieval of the number of troops for player 1	setUp()	players	numPlayers = 1	Returns "Troops of the player: [number of troops]"
Verify retrieval of the number of troops for an invalid player	setUp()	players	numPlayer = -1	Returns "Troops of the player: [number of troops]"

Test Objective	Scenery	Configuration	Inputs	Expected Result

Verify generation of militar strategy and deduction of troops for player 0	setUp()	isMatrix	numPlayers = 0	Returns the militar strategy and deducts troops from player 0
Verify generation of militar strategy and deduction of troops for player 1	setUp()	isMatrix	numPlayers = 1	Returns the militar strategy and deducts troops from player 1
Verify insufficient troops for generating militar strategy for player 0	setUp()	isMatrix	numPlayers = 0	Returns "You don't have enough troops"

Test Objective	**Scenery**	**Configuration**	**Inputs**	**Expected Result**

Verify string representation of territories using matrix	setUp()	isMatrix	N/A	Returns the string representation of territories using the matrix
Verify string representation of territories using list	setUp()	!isMatrix	N/A	Returns the string representation of territories using the adjacency list
Verify empty string representation of territories when no territories exist	setUpEmpty()	isMatrix	N/A	Returns an empty string representation

Test Objective	**Scenery**	**name1**	**name2**	**Expected Result**
Verify setting names of players	setUp()	"John"	"Alice"	The names of the players are set as "John" and "Alice"

Verify setting empty names	setUp()	""	""	The names of the players are set as empty strings
Verify setting long names	setUp()	"LongName"	"VeryLongName"	The names of the players are set as "LongName" and "VeryLongName"

Test Objective	**Scenery**	**Configuration**	**Inputs**	**Expected Result**
Verify setting territories and scores for players in adjacency list	setUp()	isMatrix	N/A	Territories and scores are set for players using adjacency list
Verify setting territories and scores for players in matrix	setUp()	isMatrix	N/A	Territories and scores are set for players using matrix

Verify correct assignment of territories and scores for players	setUp()	isMatrix	N/A	Players have distinct territories and scores assigned

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify printing of territories conquered by player 0	setUp()	players	player = 0	Returns a string with the territories conquered by player 0
Verify printing of territories conquered by player 1	setUp()	players	player = 1	Returns a string with the territories conquered by player 1
Verify printing of territories conquered by an invalid player	setUp()	players	player = -1	Returns an empty string

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify retrieval of the current territory for player 0	setUp()	players	player = 0	Returns the current territory of player 0
Verify retrieval of the current territory for player 1	setUp()	players	player = 1	Returns the current territory of player 1
Verify retrieval of the current territory for an invalid player	setUp()	players	player = -1	Returns null

Test Objective	Scenery	Configuration	Inputs	Expected Result

Verify printing of adjacent territories and their required troops (List)	setUp()	isMatrix	player = 0	Returns a formatted string listing the adjacent territories and their required troops for player 0
Verify printing of adjacent territories and their required troops (Matrix)	setUp()	isMatrix	player = 1	Returns a formatted string listing the adjacent territories and their required troops for player 1
Verify printing when there are no available territories to conquer	setUp()	isMatrix	player = 0	Returns a string indicating that there are no available territories to conquer in the current territory of player 0

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify printing of minimum path between two territories (List)	setUp()	isMatrix	territoryName = "A", territoryDestination = "D"	Returns a formatted string listing the minimum path between territory "A" and "D" and the cost of troops required
Verify printing of minimum path between two territories (Matrix)	setUp()	isMatrix	territoryName = "A", territoryDestination = "E"	Returns a formatted string listing the minimum path between territory "A" and "E" and the cost of troops required

Verify printing when the starting territory doesn't exist	setUp()	isMatrix	territoryName = "X", territoryDestination = "D"	Returns a string indicating that the starting territory doesn't exist or is misspelled

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify reconstruction of path from start to destination vertex	setUp()		previousVertices, startVertexValue, destinationVertexValue	Returns a list of vertices representing the reconstructed path from the start vertex to the destination vertex, including the total weight

Verify reconstruction of path when there is no valid path	setUp()		previousVertices, startVertexValue, destinationVertexValue	Returns an empty list when there is no valid path from the start vertex to the destination vertex
Verify reconstruction of path when start and destination vertices are the same	setUp()		previousVertices, startVertexValue, startVertexValue	Returns a list with a single element containing the start vertex followed by the total weight

Test Objective	**Scenery**	**Configuration**	**Inputs**	**Expected Result**
Verify addition of edges to graphs	setUp()	managerPersistence, graphAdjacencyList, graphMatrix	edges array with valid data	Edges are added to both graphAdjacencyList and graphMatrix

Verify addition of edges with invalid data to graphs	setUp()	managerPersistence, graphAdjacencyList, graphMatrix	edges array with invalid data	No edges are added to graphAdjacencyList and graphMatrix
Verify addition of edges with empty data to graphs	setUp()	managerPersistence, graphAdjacencyList, graphMatrix	edges array with empty data	No edges are added to graphAdjacencyList and graphMatrix

Test Objective	**Scenery**	**Configuration**	**Inputs**	**Expected Result**
Verify calculation and comparison of scores	setUp()	players	-	Returns the winner and scores based on player scores
Verify winner and scores with equal scores	setUp()	players	Set both player scores to the same value	Returns a draw message with both players' scores

Verify winner and scores with negative scores	setUp()	players	Set player scores to negative values	Returns the winner and scores based on player scores, considering negative values

Test Objective	Scenery	Configuration	Inputs	Expected Result
Verify winner and score when player 0 gives up	setUp()	players	player = 0	Returns the winner as player 1 and displays the score of player 1
Verify winner and score when player 1 gives up	setUp()	players	player = 1	Returns the winner as player 0 and displays the score of player 0

Test Objective	Scenery	Configuration	Inputs	Expected Result

Verify moving to a conquered territory	setUp()	players	territory = "B"	Returns a success message indicating the updated territory of the player
Verify error message when moving to un-conquered territory	setUp()	players	territory = "C"	Returns an error message indicating that the territory is not conquered by the player
Verify error message when moving to the same territory	setUp()	players	territory = "A"	Returns an error message indicating that the player is already in the same territory