DATA STRUCTURES AND ALGORITHMS II 2023-2024

Infestation at UPF - The Game

Mauro Bruera Méndez, Nour Esbrí Miñambres, Jana Baguer Fabrega
U232863, U213333, U213970

29/05/2024

Table of Contents

1. Introduction

Given the Role Playing Game prototype task, it was decided to proceed with “Infestation at UPF”. The proposed game is based at the Pompeu Fabra University at Campus Poblenou and is set in an apocalyptic future. The player finds themselves studying at the Library. After the first setting, the user starts with some basic decisions that will affect the course of action of the game.

The developed game consists of four different scenarios: the Library, “Tallers”, Café and Classroom. Additionally, a transitional scenario is placed between the second and third, Gutenberg Square, where the player makes a choice that affects their statistics. In each scenario - with the exception of Gutenberg Square - there are three sub-scenarios where the player faces certain enemies according to their previous choices. Once the player wins a battle, they are prompted with the option of travelling to other sub-scenarios or continuing with the main storyline. Throughout the course of the game, the player encounters certain items that affect their statistics.

Although no existing solutions were used for inspiration, external sources were implemented so as to enable the functionality of “JSON” (JSON.org, 2024). The use of the aforementioned sources is tackled in the ethical implications of the report.

Regarding the project’s objectives, all of the mandatory objectives have been met. Furthermore, of the desirables the first, second, third and fourth have been met. Lastly, of the exploratory the first and fifth have been met.
The following report delves into the dynamics of “Infestation at UPF”, the project’s objectives developed, the process of elaboration, and the ethical implications.

2. Project Objectives

- **Mandatory Objective Net**

1. Creation of Character and Other Structs

The structure for the main character is found in "game_data_structures.h". There, it is assigned a name, point values to determine the player's statistics in fights - health (hp), attack (atp), defence (def) -, and a degree to determine the initial statistics. Also a number of skills, the skills array to determine the player's choices of skills and the stack node to keep track of move history - which are separate structures -, and a flag for the time strike move.

The implementation is found in the "menu_config.c" file, where the initial menu of the game is coded. This consists of three options; starting a new game, configuring the player's character, and exiting the game. The second option is where the player will be prompted to insert a name - which is assigned to the initialised character structure - and to choose a degree; the latter will determine the initial points of hp, atp, and def. The final configuration is displayed once all mentioned prompts are covered.

In terms of the code, through line 30 to line 82 a switch-case algorithm was chosen to determine the aforementioned depending on "degree_choice" previously prompted to the player. Its behaviour is expected to go through the cases and execute that which matches the integer number selected for "degree_choice". In case of lack of match, the algorithm is programmed to automatically diverge to the default setting which is equivalent to case 3. Regarding the Big O, the switch-case algorithm implemented uses consecutive cases (1, 2, 3…) which is O(1) from the jump table.

This algorithm could be enhanced by implementing an error-checking algorithm that prompts the user again if their choice is not matched to any case.

On the other hand, other structures are defined in the "game_data_structures.h" file:

- Attributes: Three integer value variables (hp, atp, def) to store the statistics respective values. Used for skills and subsequently enemies and character.

- Skill:
    - Name: character array, size 20.
    - Description: character array, size 200.
    - Type: boolean value. If true, direct attack. If false, temporary modifier.
    - Duration: integer value. Determines the number of turns the skill lasts for. If it is a direct (permanent) attack, the value is set to -1.
    - Attributes: modifier (aforementioned structure to store statistics' modifications of the skill).
    - Active: boolean value. Determines whether the temporary modifier has been used in combat but is not exhausted.
- Enemy:
    - Name: character array, size 50.
    - Description: character array, size 250.
    - Points (hp, atp, def): integer values to directly store the enemy's statistics.
    - Skill: array of skills (aforementioned structure) of size "MAX_SKILLS" which is a macro defined in game_data_structures.h with value 4. Stores the enemy's predetermined attacks.
- Option:
    - Response: character array, size 100. Stores the response to the player's choice.
    - Narrative Before: character array, size 600. Stores the narrative previous to a battle.
    - Enemies: array of enemies (aforementioned structure), size "MAX_ENEMIES" which is a macro defined in game_data_structures.h with value 11. Stores the enemies the player will face.
    - Narrative After: character array, size 600. Stores the narrative post-battle.
- Scenario:
    - Name: character array, size 50.
    - Description: character array, size 1000.
    - Decision Options: array of option (aforementioned structure), size "MAX_OPTIONS" which is a macro defined in game_data_structures.h with value 3. Stores available sub-scenario options (places of battle).
- Stack Node:

    - Skill: skill (aforementioned structure). Stores the skill’s information saved in the stack.
    - Next: pointer to “next” Stack Node. Points to the next node in the stack.
- Skill Usage:
    - Skill Name: character array, size 50. Stores respective skill’s name.
    - Usage Count: integer value. Stores the number of times it has been used.
- Skill Dictionary:
    - Skills: array of Skill Usage (aforementioned structure), size “MAX_SKILLS” (aforementioned macro).
    - Size: integer value. Stores the size of the dictionary.
- Turn Node:
    - Next: pointer to “next” Turn Node. Points to the next turn (node).
    - Type: character value; “e” for enemy, “p” for player. To determine who’s turn it is.
    - En Index: integer value. Stores the index of the position of an enemy in the array scenario_enemies.
- Turn Queue:
    - Front: pointer to the front of the queue, type Turn Node (aforementioned structure).
    - Rear: pointer to the rear of the queue, type Turn Node (aforementioned structure).
    - Number of Enemies: integer values.
- Scenario Node:
    - Scenario: information about the scenario, type Scenario (aforementioned structure).
    - Next: pointer to “next” Scenario Node. Points to the next node in the graph.
- Scenario Graph:
    - Nodes: array of Scenario Node (aforementioned structure), size “MAX_SCENARIOS” defined in game_data_structures.h with value 50. Stores the nodes representing scenarios.
    - Number of Nodes: integer value. Stores the number of nodes in the graph.

This objective was developed in 8 hours approximately, mainly due to various changes it received throughout the game’s elaboration.

2. Scenario’s Texts and Battles

The data of the scenarios is stored and loaded from a JSON file named “game_config.json” through the function defined in “load_game_data.c”. The scenario is initialised in its respective structure which stores a response, a narrative previous and posterior to the battle, as well as the decision options the player can choose from which will determine the sub-scenario it travels to and the enemies it faces. These sub-scenarios have their own response and previous narrative to the battle as well. Once the battle is won by the player, they will be prompted with the option of travelling to another of the adjacent sub-scenarios to face other enemies. In fact, they will not be able to move further on in the storyline until they visit a minimum of 2 sub-scenarios within their current scenario.

To implement this objective in the file “menu_config.c”:

- Scenario Initialise (scenario_initialise): the algorithm used here is a for-loop that prints the different decision options found in this scenario so as to have the player choose one and then call the next function. Additionally, this function prints the information about the current scenario and displays the available skills the player can choose from, and will prompt them to choose 4 by calling the function “choose_skills”.
- Sub-scenario Initialise (subscenario_initialise): this function uses the algorithm of for-loop as well, various times, to determine the number of enemies, as well as to initialise them in an array that will then be passed on to the “fight_flow” function (delved into further on).

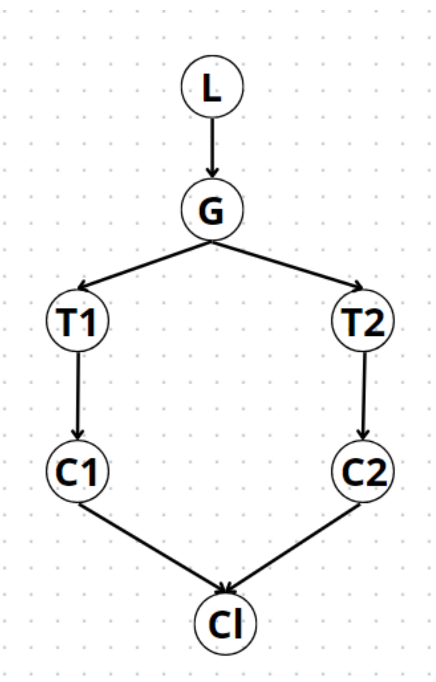
For “scenario_initialise”, the time execution is of O(n) since all for-loops used are “single” loops. For “subscenario_initialise”, it will be of $O(n^2)$ since the for-loop tackled is a loop with an inner loop. It must be noted that the loops used for error-checking and enemy initialisation are of time O(n) since they are “single” loops.

As a limitation, the algorithms depend on the numerous variables their respective functions take, which can sometimes lead to misconnections or lack of information.

The implementation of this objective is found in the “menu_config.c” file, through lines 121 to 224. The approximate time to develop has been 10 hours, mainly due to changes over the course of elaboration.

3. Graph Creation

The graph system is implemented throughout graph_scenario.c, game_data_structures.h, fight_flow.c, game_test.c, menu_config.c and load_game_data.c. Since there are scenarios to which the player cannot move until they have passed through other scenarios first, we have decided to use a tree for the graph (to not implement a graph with conditional nodes). The scenarios are: Library(L), Gutenberg Square(G), Tallers(T1/T2), and Café(C1/C2).



The information of T1 is the same as of T2, and the same applies to C1 and C2.

To create the graph we defined two structs in game_data_structes.h: ScenarioNode and ScenarioGraph. The graph is declared in game_test.c. Then, in load_game_data.c, with the function load_game_data(...) we partially populate the array of pointers of ScenarioGraph. To do that, we create a ScenarioNode calling createScenarioNode(*Scenario scenario). Finally, we add this ScenarioNode to the array in ScenarioGraph calling addScenarioNode(...). In case the scenario is either Tallers or Cafeteria, we duplicate them. This is because there are two possible routes for the player when he is at Gutenberg Square, meaning that T1 has different neighbours than T2.

Until now, ScenarioGraph only includes the array of ScenarioNodes (graph_nodes = [L, G, T1, T2, C1, C2, CL]) and the number of nodes in the graph (updated in addScenarioNode(...)). However, the array of neighbours of each ScenarioNode is not initialised. Hence, in load_game_data(...) we call connectScenarioNodes(ScenarioGraph* graph). This function fills this array with the corresponding neighbours. The neighbours variable is a pointer. Hence, since Classroom is the end node, the function makes neighbours point to NULL. At the end of the main function, we use freeScenarioGraph.

In the function fight_flow_management(...) we use the graph to move to the next scenario after the fight. Furthermore,in menu_config.c, the function start_new_game(...) creates a variable 'ScenarioNode* current_node', which initially points to the first node of the graph (L), and then calls the function subscenario_initialise(...), passing current_node as a parameter. subscenario_initialise(...) calls fight_flow_management(...) also passing current_node as a parameter. If the player wants to change the scenario, we update current_node such that it now points to the next neighbour.

The most expensive function is freeScenarioGraph, which has an O(n). The others have O(1), as well as the step of updating the current node. Overall, the step that consumes more time is the loading of data into each scenario that we then transform into a ScenarioNode. Because of the nested loops, it results in an $0(n^4)$ – polynomial time performance.

An improvement could be to include the sub-scenarios of each scenario into the graph (right now they are accessed as options and nothing more). Also, as we have programmed the game, the player cannot go back to any scenario. We could make the graph more flexible in that sense.

We have spent approximately 8 to 10 hours implementing the graph system and ensuring it was well integrated into the game.

4. Battle System

The battle system is implemented in two files, queue_turn_system.c and fight_flow.c. The first file uses the structures TurnNode and TurnQueue, and defines the functions:

- createQueue(...). O(1)
- isQueueEmpty(...). O(1)
- enqueue(...). O(1)
- dequeue(...). Returns the pointer to the character (enemy or player) dequeued. O(1)
- initialiseBattleQueue(...). This function will be explained in objective 7.

These functions are used to create the queue of turns of a fight between the player and the enemies. This basic structure is then used in the function that deals with the loop of the fight: fight_flow(...).

The loop of the fight continues as long as the queue is not empty. In each turn, a character is dequeued and stored as currentCharacter.

If it is a player, then processPlayerTurn is called. This function displays the players' available skills (skills_menu(...)), makes them choose and deals with the selection (skills_handling(...)). skill_handling(...) updates the player's statistics and the duration of modifiers. Also, it implements necessary changes to prevent consumed temporary modifiers from appearing in the menu in the next player's turn. Then, processPlayerTurn computes the damage with calculateDamage, and allows the player to choose an enemy to attack. fight_flow(...) checks whether all enemies are dead via enemies_defeated(...).

On the other hand, if it's an enemy's turn, the function calls processPlayerTurn(...). A random skill is selected and handled, and following the previous process, the enemy attacks the player. After that, if the player is dead, fight_flow(...) returns false. To sum up:

- calculateDamage O(1)
- enemies_defeated(...). O(n)
- skills_menu(...). O(n)
- skill_handling(...). O(n)
- processPlayerTurn(...). O(n*m) – highest of all nested functions.
- processEnemyTurn(...). O(1)
- fight_flow(...). O(n*m) at worst – if processing processPlayerTurn(...). And O(1) at best.

Then, fight_flow_management(...) deals with the result of the fight(true/false) and links the graph system with the battle_system. If false, the game is over, if true, the player can decide whether to continue with the story, or to fight in another subscenario (exploratory 2). It is important to highlight that if the player is in G, they must choose between T1 or C2 and then update the current node:

- Nodes with one neighbour:

current_node current_node->neighbours[0]

- GutenbergSquare:
  current_node = current_node->neighbours[0]->neighbours[choice-1]
- End node:
  In menu_config.c, the function subscenario_initialise(...) is programmed that after one fight in Classroom, the player has won.

This section's development time was approximately 30h.

5. Enemy Assignment and Configuration

The structure of the enemy is defined in "game_data_structures.h" and in the JSON file, each sub-scenario has an enemy assigned with their statistics and same set of skills. The enemies are initialised in "subscenario_initialise" at "menu_config.c" as well as an array of the sub-scenario's enemies, and an integer variable named "num_enemies". Thereafter, an array of type pointer to "struct Character" is initialised with length "MAX_ENEMIES" defined in "game_data_structures.h", named "scenario_enemies". A for-loop then will iterate through the chosen sub-scenario's enemies (defined in JSON) and will assign each value of each variable to the new previously defined array of enemies. The enemy's name, "hp", "atp", "def", and skills will be assigned, and the degree of the enemy - as it is an array pointer of type character - is set as "enemy".

Once the for-loop iterates through all enemies in the chosen sub-scenario, the "fight_flow" function is called to start the battle of the game. Once it is finished, the array of enemies is freed.

The different types of enemy are defined according to their respective statistics. As the player keeps on playing further on in the game, these values increase so as to make the battles more difficult. Once the battle begins and it is the enemy's turn, a random skill of the available is implemented.

The main function to initialise the enemies and assign their corresponding values is "subscenario_initialise". Specifically the lines 197 to 240 in the file "menu_config.c". Since this function uses a nested for-loop with two loops with execution time O(n^2). Additionally, for each if-else structure in the function, there will be an O(n). For the section where the if-else contains a double nested loop, the time execution is O(n^3).

To fully develop this objective, it took approximately 8 hours to define all functions, configurations and dynamics for the enemies' assignment and configuration.

6. Time Strike

To complete this objective, multiple variables were used. These include player which represents the character; dictionary, which tracks the usage of each skill; top, which is a pointer to the top of the stack and manages move history; newNode, which represents a new node in the stack; temp, which temporarily holds the node being popped form the stack; current, that traverses the stack to count moves or find a specific move; and many others. Then, there is StackNode, that stores the history of moves in a stack format. Therefore, it allows LIFO access to moves and the different operations related to it (*push*, *pop*, *add* and *remove*). Moreover SkillDictionary is a structure that tracks the usage count of each skill.

The functions used for this target are all found in the file time_strike.c and are the following:

- push(...): this function adds a skill to the top of the stack and updates the skill dictionary. It is called every time a skill is about to be used and the algorithm is based upon creating a new node, and inserting it at the top of the stack. Furthermore, it has a performance of O(1).

- pop(...): the purpose of this function is to remove and return a skill from the top of the stack. Once the top node is removed, that node's memory is also freed. It has a performance of O(1).

- timeStrike(...): this is the main function of the implementation as it executes the "Time Strike" move. The algorithm consists of checking whether this functionality has been already used and, if not, count the number of moves in the history stack. Then, to randomly select a move from the history, double its power and re-execute it. Finally, the "Time Strike" is marked as already used and it returns the index of the Skill according to the array of skills of the player. Consequently, we are dealing with a performance of

O(n) for counting moves and O(n*m) for finding the skill in the array (where *n* is the number of moves in history and *m* is the maximum number of skills).

- getNthFromTop(...): this function retrieves the index of the n-th skill from the top of the stack in the player's skill array once one has been randomly selected during the "Time Strike". Hence, it is based upon traversing the stack to the n-th node and then finding the index of the skill in the player's skill array. Because of this, its performance is O(n*m) where *n* is the number of nodes to traverse and *m* is the number of skills.

In terms of error handling and limitations, the function getNthFromTop returns -1 if the skill is not found, indicating an error. And, according to the skills created, the algorithm assumes they have unique names in the player's skill array. Finally, the performance can degrade with a large number of moves due to repeated traversal of the stack. However, the game is limited and relatively short.

Ultimately, some possible improvements include optimising the search mechanism in getNthFromTop for better performance or adding more robust error handling to manage edge cases.

This objective was developed in approximately six hours due to errors encountered. It is found mainly in the file time_strike.c and time_strike.h, apart from the definition of the data structure which is found along with the others at game_data_structures.h.

7. Game Turns

The game turns are defined in the function initializeBattleQueue(...). This function chooses a random number (0,1) and identifies it with the player. Then, in a for loop it keeps enqueuing the player and the enemy until the maximum number of turns are reached. Because of the condition 'i % 2 == playerTurn', we ensure that in each battle, whether the player starts or not is random.

The only variable used is"playerTurn" which, as mentioned, randomly determines the first attacker.

We could improve this function by making the code more flexible. Since, in the end, we only have one enemy for sub-scenario we simply enqueued one enemy. If instead we left the commented for loop, our program could be used for sub-scenarios with multiple enemies. It was our initial intention (as can be observed in other sections of the code), but we reduced the number of enemies for simplicity.

The time complexity of this function is O(n), and it took approximately 20 minutes to program it,

- **Desirable Objective Net**

1. Load Data from JSON file

As mentioned above, the data for the scenarios, decision options, enemies and skills is loaded from a JSON file named "game_config.json". This file follows the JSON structure through two main branches; scenarios and player skills. Within the scenarios, all information regarding each scenario - its corresponding decision options (sub-scenarios), and within these their assigned enemies, with their respective enemy skills as well - is stored following the data structures defined in "game_data_structures.h". For the player skills, the same storing structure is used, though not as much information needs to be stored; each player skill contains the corresponding information and its attributes, as defined in the structure of "game_data_structures.h".

The reason for this separation between scenarios-decisions-enemies and player skills is that these two sets of data are used and managed differently. The structure for the scenarios is designed so as to be able to navigate through it with all parameters determined and defined, while allowing obtaining the needed data depending on the player's choices. However, the player skills are constant and loaded from JSON every time a scenario is initialised, therefore the structure is simpler so as to ensure the maintenance of the data as well as the individual management of it, independent of the player's choices regarding scenarios.

In order to open, read, and parse the JSON file, as well as extracting all of the data and initialising the corresponding variables for each structure, the function "load_game_data" defined in the file "load_game_data.c" through lines 5 to 214 is implemented. This function uses the syntax and functions from cJSON, found in the file "cJSON.c" which was obtained from the public git repository "DaveGamble/cJSON" (Gamble D., 2024). This function is called in the main file of the game, "game_test.c", in order to load all of the data so as to use it throughout the game without the need to access JSON constantly.

Various algorithms are used in this function. In order to synthesise, the following are tackled:

- For Loop (tripe): in the case of scenarios, decision options, and enemies, a triple for loop is used; that is, two for loops within a for loop. Since the algorithm will execute through O(n*n*n), the execution time will be O(n^3).
- For Loop (single): for single loops, as seen before, the execution time will be O(n).
- If, Else: for error-checking, the if-else statement is used. This algorithm has an execution time of O(n).
- If: for single error-checking - that is, checking if items are empty or not found - an if statement is used. This algorithm has an execution time of O(1).

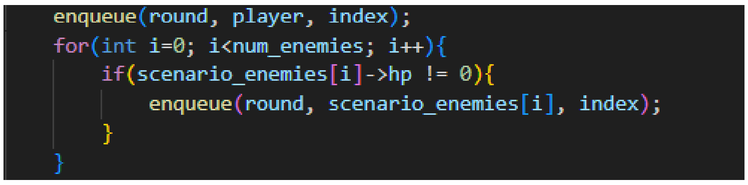
This function will go over the whole JSON file, "game_config.json", and will extract the data and initialise it with the defined structures.

The development of this objective was done through approximately 12 hours of work, mainly due to the changes made throughout the elaboration, as well as the initial complexity it entailed for the use of cJSON.

2. Time Execution Analysis of 2 Functionalities

We are going to analyse the time performance of 2 applications of a DataStructure/Algorithm to develop a functionality of the game.

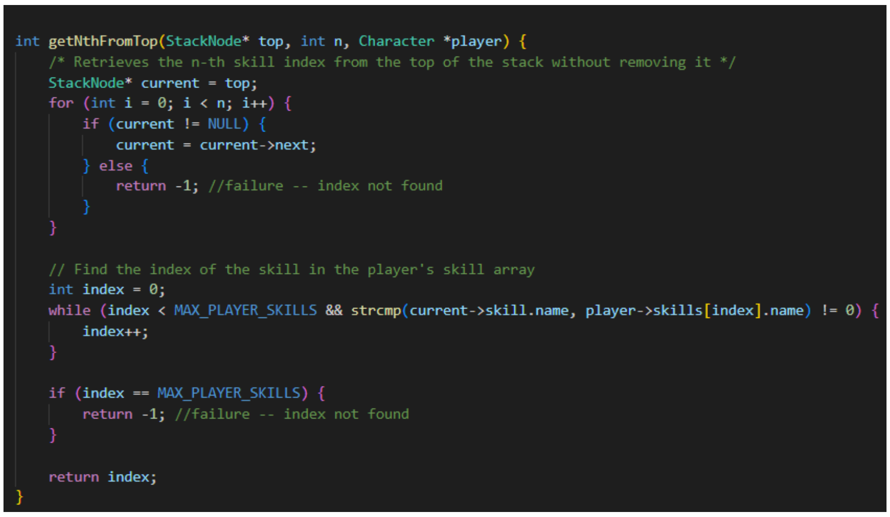
1) Filling a round



```
enqueue(round, player, index);
for(int i=0; i<num_enemies; i++){
    if(scenario_enemies[i]->hp != 0){
        enqueue(round, scenario_enemies[i], index);
    }
}
```

As previously noted, enqueuing takes constant time, so the first line takes O(1). In the worst case, all enemies would be alive, so the code performs n=num_enemies times an enqueue, which takes O(1). Hence, we have: O(1)+O(1)*num_enemies = O(num_enemies). Filling a round takes linear time.

2)GetNthPosition



```c
int getNthFromTop(StackNode* top, int n, Character *player) {
    /* Retrieves the n-th skill index from the top of the stack without removing it */
    StackNode* current = top;
    for (int i = 0; i < n; i++) {
        if (current != NULL) {
            current = current->next;
        } else {
            return -1; //failure -- index not found
        }
    }

    // Find the index of the skill in the player's skill array
    int index = 0;
    while (index < MAX_PLAYER_SKILLS && strcmp(current->skill.name, player->skills[index].name) != 0) {
        index++;
    }

    if (index == MAX_PLAYER_SKILLS) {
        return -1; //failure -- index not found
    }

    return index;
}
```

The time complexity of the first for loop depends on the number of times the if statement is called (n times), as well as the operation within the conditional statement, which takes O(1). Hence, the first loop takes n*O(1) = O(n). The while loop runs at most MAX_PLAYER_SKILLS times, so it takes O(MAX_PLAYER_SKILLS). Since the last conditional is constant, the function that defines the complexity is, by the rule of addition of complexities: O(n)+O(MAX_PLAYER_SKILLS)+O(1). By the rule of dropping constant terms, and the rule of focusing on dominant terms, the final Big O should be O(MAX_PLAYER_SKILLS) (MAX_PLAYER_SKILLS will always be greater or equal to n).

3. Dictionary Structure for Usage Count

The objective was to implement a “SkillDictionary” that manages the skills available to the player in the game. This involved creating a data structure to store skill information and an algorithm to initialise and utilise this dictionary efficiently during the game.

The variables used are SkillDictionary* dictionary which is a pointer to the structure that holds all available skills (SkillDictionary). Skill *playerSkills, which is an array of Skill structures representing the skills that the player can use. And, finally, Character *player which is the player character whose skills are being managed through the SkillDictionary.

Hence, the chosen data structures include SkillDictionary and Skill. The first one holds an array of Skill objects and manages their initialization and retrieval. Furthermore, it is designed to provide fast access to skills and their properties. The later one is not unique and was previously explained in Mandatory 1.

The functions used for this target are all found in the file time_strike.c. Apart from those functions stated at the Mandatory 6, these are specific of this section:

- initializeSkillDictionary(...): this function initialises the SkillDictionary with the provided skills array up to a maximum number of skills. This has a time complexity of O(n) where *n* is the number of skills being initialised. This is because each skill is copied into the dictionary once.

- updateSkillUsage(...): this function iterates through the SkillDictionary to find a skill by name. If the skill is found, its usage_count is incremented by 1. Otherwise, an error message is displayed as all skill names have been initialised in the dictionary. The said algorithm has a time complexity of O(n) in the worst case, where *n* is the number of skills currently in the dictionary.

There are some algorithmic limitations. Such as the fact that the search for a skill is linear, making it less efficient for larger dictionaries. However, since the game is limited as well as the number of skills that can be saved, it was decided to proceed with a linear search. Furthermore, the dictionary has a fixed maximum size which can lead to issues if the number of skills exceeds this limit. Nevertheless, we decided upon a value that would not cause problems in that sense.

Consequently, some improvements would be to use a hash table for faster look-up times, reducing the complexity to O(1) on average. Also, to do dynamic resizing which would allow the dictionary to resize dynamically to handle more skills as needed.

Finally, the implementation of the SkillDicitonary is found in the time_strike.c in terms of definition of functions and in the fight_flow.c as well as in menu_config.c for its initialization and calling. Furthermore, the structure is defined in the game_data_structures.h header file.

This objective was developed in approximately two hours, as it was mostly related to linking it with the Mandatory 6. Furthermore, we found some issues

4. Implementation of a Search/Sorting Algorithm

For the implementation of the function fight_flow_management(...) we have used an auxiliary function, is_index(). This function checks whether an integer is within an array. Since the length of the array is 3 (short), we have used the linear search algorithm instead of the binary search. This function is used to know whether a sub-scenario has already been visited, since the array we pass is index_scenarios[MAX_OPTIONS]. If the function returns true, it means that the player has already been there, so in the option's display, this sub-scenario won't be shown as an option.

The time complexity of this algorithm is O(n), but since the length of the array is so short, it is fast.

This objective took approximately 10 minutes, since it is a straightforward basic algorithm seen in class.

- **Exploratory Objective Net**

1. UX/UI Improvements

Since the game developed for this project is terminal-output based, it was decided to enhance the player's experience through embellishments to improve the UX/UI. As suggested by the lab professor external sources such as OpenAI's ChatGPT-4o were used (OpenAI, 2024).

To improve the interface, ANSI escapes were used to bring different colours to both the front and backgrounds. The implementation can be seen throughout the code in most files, with special concentration in menu-display functions such as “menu” or “configure_character” in the file “menu_config.c”.

The main concepts used to improve the UX/UI are:

- ANSI Escapes: As shown in the referenced GPT chat, these standardised codes allow for the developers to set the display look in the console/terminal.

- Unicode: Similarly to the previous standardised codes, Unicode is a format in which the developer can introduce “emojis” into their program display by typing in the corresponding codes.
- Figlet: This computer program was installed externally to the used platform to produce text banners composed by conglomerations of ASCII characters. Though the installation of Figlet was assisted by ChatGPT as seen on the reference, its use and the production of the banners was made entirely in the computer’s terminal with Figlet.

To fully implement the enhancements made for the interface, it took approximately 10 hours. Most of that given time was spent on research, ideas, and, mainly, the incorporation of the improvements all over the code

2. Other Suggestions

The developers agreed on imposing the condition that a player cannot continue with the story unless they have fought twice in a scenario. This has been implemented in the function fight_flow_management(...). To summarise:

- If the player has held less than two battles, they are forced into choosing another sub-scenario of the current scenario.
- If the player has held three fights, they are forced to move to the next scenario.
- If the player has held two fights, they can choose between fighting again or continuing with the story.
- To do this, we use the variable int* fights_scenario and the array int index_scenarios[MAX_OPTIONS]. To keep track of the battles that have occurred in a scenario and the subscenarios visited. This is so that the menu of options only displays the non-visited sub-scenarios.

Checking the number of fights takes constant time, while checking whether a scenario has been visited takes linear time.

The reasoning and programming of this particular condition took approximately 2 hours.

3. Solution

- **System Architecture**

The architecture of the project is organised into several key components that work together to manage the game's flow, including character configuration, scenario progression, and skill management.

(1) Game Initialization: loads scenarios from a configuration file or data structure and initialises the scenario graph. Furthermore, it sets up the player character's initial state.
(2) Character Configuration: handles the configuration and customization of the player's character. Moreover, it prompts the player for input and sets attributes such as name, degree, health ponts (HP), attack points (ATP), and defence points (DEF).
(3) Scenario Management: manages the flow of the game scenarios as well as directs the game through different scenarios based on the user's choices. Consequently, it also manages sub-scenarios and tracks the player's progress through the game.
(4) Skill Management: manages the player's skills and allows players to select skills for each scenario. Also, it updates the skill dictionary based on skill usage and maintains a record of skill usage counts.
(5) Combat System: handles the combat mechanics within the game, manages the turn-based fighting system, calculates damage and controls the sequence of attacks between the player and enemies.
(6) Error Handling and Logging: ensure the errors are managed gracefully and logged appropriately. Furthermore, it validates input, hangles unexpected scenarios, and provides feedback to the player. Finally, it logs errors for debugging and system improvement.

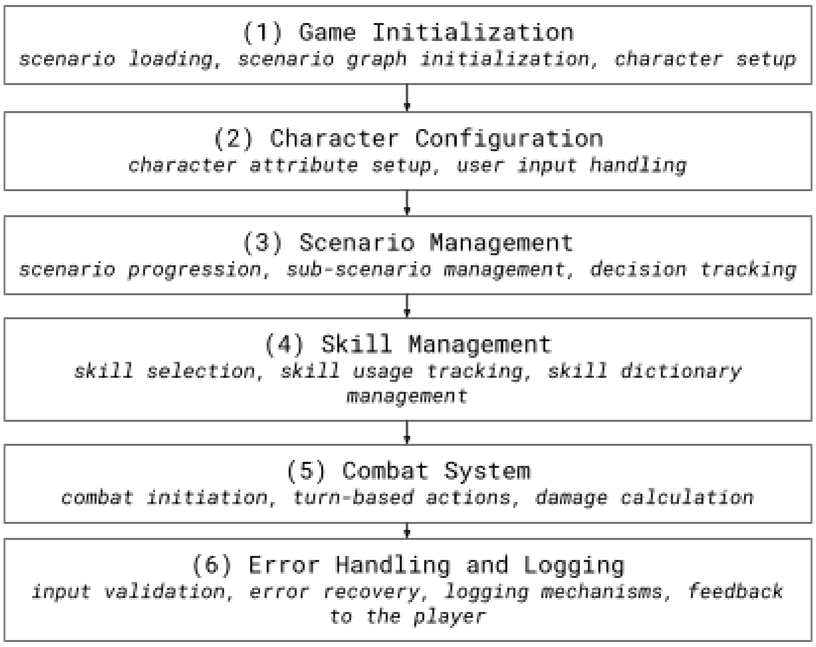

(1) Game Initialization
scenario loading, scenario graph initialization, character setup
(2) Character Configuration
character attribute setup, user input handling
(3) Scenario Management
scenario progression, sub-scenario management, decision tracking
(4) Skill Management
skill selection, skill usage tracking, skill dictionary management
(5) Combat System
combat initiation, turn-based actions, damage calculation
(6) Error Handling and Logging
input validation, error recovery, logging mechanisms, feedback to the player

Figure 1: System Architecture Diagram

- **Error Handling**

In the project, error handling is implemented to ensure robustness as well as a smooth user experience based on several principles.

Firstly, there is input validation to prevent invalid data from being processed. For instance, when configuring the character or choosing options in scenarios, the input is checked for validity before proceeding with the action.

Secondly, the system is designed to handle errors gracefully. When an error occurs the system provides a meaningful error message to the user and attempts to recover or exit safely. This was mainly implemented to make the development of the game easier.

Then, there is the logging error handling. When an error occurs during initialization of the combat system, for instance, a message is logged indicating the nature of the error.

Finally, there is the default handling when encountering unexpected situations. The system uses mechanisms to ensure continuity. To illustrate, if an invalid degree choice is made during character configuration, the system defaults to a predefined degree, which in our case is the Computer Science one.

- **Data Model Design**

The data model design for this project revolves around several key structures that encapsulate the game's entities and their interactions.

Firstly, there is the Character Configuration. Where the user inputs are captured and stored in the Character structure.

Later, there is Skill Management, where skills chosen by the player are stored in the Character structure as well and their usage is tracked via the SkillDictionary.

On the other hand, Scenario Progression allows the proper game flow of moves through different scenarios as defined in ScenarioGraph, given by user input. Each scenario may involve combat, skill usage, and further decisions affecting the game state.

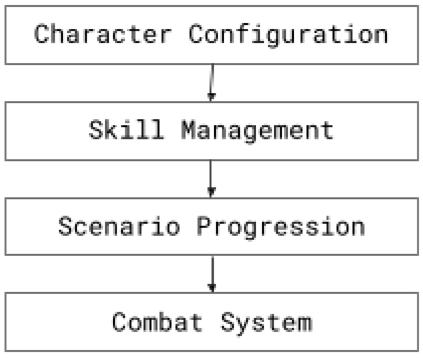


Figure 2: Data Flow

Lastly, the Combat System appears during each combat. The characters' attributes and skills are used to calculate damage and outcomes and it is given by the TunQueue, which manages the sequence of actions.

- **Dataset Description & Processing**

The dataset for this project consists of predefined scenarios, skills, and characters – enemies to be more specific. These datasets are read and processed at the start of the game to initialise the game environment.

(1) Scenarios: there are the descriptions of various game scenarios, including narrative text to introduce the user to that world. Also, decision options, and potential enemies according to those.

(2) Skills: the skills are described and available to the player, including the effects they have in the Character's characteristics and usage count.

(3) Characters: there are predefined attributes for both players and enemies characters. These include the name, degree, HP, ATP, and DEF data.

The processing of the discussed datasets is dependent on each type. For instance, the scenario initialization is based upon the reading of that data in the configuration file and then each scenario is stored in the *Scenario* structure. After that, the relationship between the multiple scenarios is represented in the *ScenarioGraph*.

A similar process is followed for the *Skill* initialization. Skills are read from the configuration file and each one is stored in the *Skill* structure. Simultaneously, the *SillDictionary* is initialised to track skill usage during the whole game.

Finally, the character initialisation is based upon the configuration the user does via input data. This includes the name and the degree. Depending on which was chosen, the HP, ATP, and DEF have different values and are stored in the *Character* structure.

4. Ethical Considerations

The work provided for the EDA II project has been written by the three authors. The main files and functions of the code, as well as the additional pieces provided such as this report and the video, were entirely produced by the aforementioned.

Additionally, regarding the use of "JSON" in the code, the authors made use of external assistance (JSON.org, 2024). In order to be acquainted with the structure and dynamics of "JSON", the Artificial Intelligence program "ChatGPT-3.5" was used as well as to access the download and activation processes needed, and to acquire ASCII-based art knowledge to improve the UI/UX (OpenAI, 2024) (OpenAI, 2024). Two of the files in the project were obtained through a public repository made to enable the use of "cJSON" named "DaveGamble/cJSON" from GitHub (Gamble D., 2024).

The use of the provided external sources was made responsibly and with the necessary acknowledgements. It must be noted that the referenced use of "ChatGPT" is not acknowledged in the presented code regarding JSON, as it was implemented in terminals external to the project's repository so as to configure the used devices' installations, though regarding ASCII-based art credit is given where due. In the case of the “DaveGamble/cJSON” repository files, proper acknowledgement and right of use is granted and notified in the respective files of the code - “cJSON.h” and “cJSON.c”.

5. References

[1] *JSON.org*. (2024). JSON. www.json.org

[2] *Configure launch.json for C/C++ debugging in Visual Studio Code*. (2024). https://code.visualstudio.com/docs/cpp/launch-json-reference

[3] Gamble, D. (2024). *DaveGamble/cJSON*. GitHub. https://github.com/DaveGamble/cJSON

[4] OpenAI (2024). chatgpt.com. https://chatgpt.com/share/6b42da20-b7e8-4361-a7e0-eae44e978fe4

[5] OpenAI (2024). chatgpt.com https://chatgpt.com/share/4c9f19fa-48a3-47c0-b1a7-f5221b0ee011