A Level Computer Science programming project

Table of Contents

Analysis

Problem identification

There already exist lots of educational games of different kinds that can give beginners in computer science a brief introduction to what computer science is, but there are very few such games for intermediate or more advanced learners. Despite the fact that most advanced learners are driven by their own motivation rather than the fun from playing games, it would still be useful for them to have some sort of visualisations of some abstract concepts, as it is often the case that some of the algorithmic ideas in computing are rather abstract and can easily deter a new learner from further study.

Having realised that, my project is going to be writing an educational game using Python and its extension library Pygame. The game will try to illustrate some advanced computing algorithms that may or may not be taught in A Level Computer Science. The idea is that some algorithmic problems all have a very interesting story, behind which lie the complicated algorithms. So, I would like to illustrate these algorithms vividly for learners and hopefully this could encourage more people to learn these algorithms, which might seem quite daunting at first.

This problem is amenable to a computational approach for two reasons. The first one is the nature of a game. As one of the aims of this project is to present computational ideas graphically and interactively, a computer would be necessary and sufficient to provide a platform for the user because it can output some game information through the monitor and receive input from the user to do the processing. Moreover, as the theme of this game is computing theory, it would make more sense if a player is playing this game on a computer. It could also make it easier for a user to implement these algorithms if he/she is sitting in front of a computer. Being able to access the source code of the game is also a good way to better understand the implementation. Compiling a book about these theories might be an alternative but would be less interactive and interesting for the learner.

Stakeholders

Despite every effort being made to ensure this game is as interesting as possible, this game is not designed for absolute beginners who want to learn more about what computer science is, the reason for which is that this game assumes basic knowledge in computing such as a tree in graph theory and it does not include introductions to these theories. If a user wants to learn more about the implementation of an algorithm, programming knowledge may be needed. However, it is still possible for a learner to get a taste of advanced algorithms, although this game is not designed to do so. In conclusion, there are two main possible types of stakeholder:

1. Those who have already obtained some basic concepts of algorithms and want to study further in this area, then this game would be a good starting point to get a feeling;
2. Teachers who might want to utilise this game in lessons while introducing relevant topics in order to add more fun to lessons. As each level in my game will centre around a key algorithm, e.g. dynamic programming, therefore this game can be played when a certain topic will be discussed in a particular lesson.

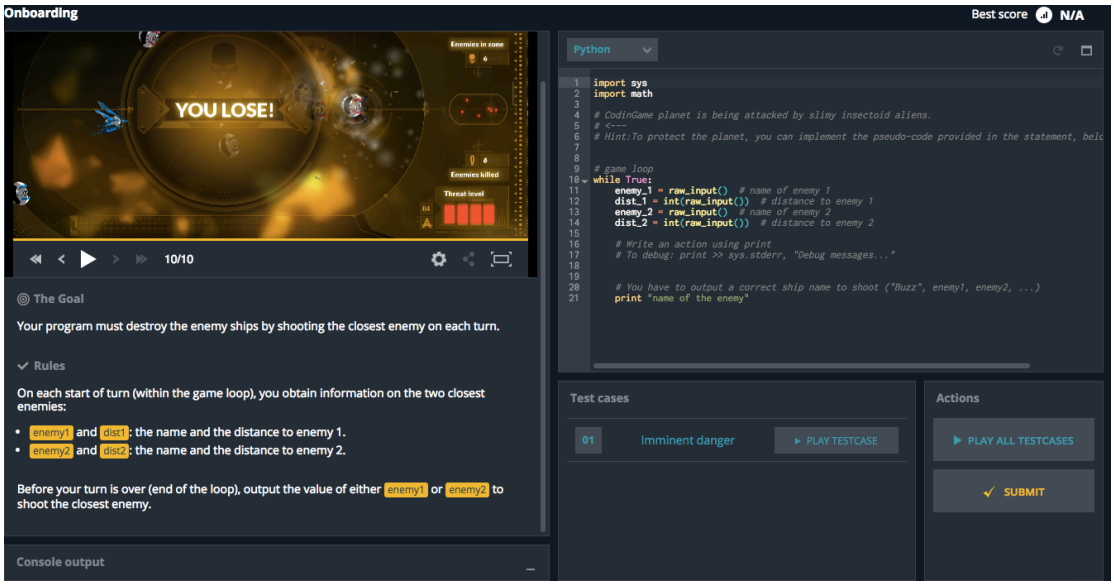
This game can be suitable to the two types of stakeholders described above for two reasons. Firstly, it does not include those most basic ideas of programming, which are not needed if users have already obtained some level of programming knowledge. Secondly, it will focus more on the visualisation and principle of the algorithms and problems, which is more important than those basic ideas such as what is a variable to those more advanced learners. As for the teaching use of this game, the teacher can give students a basic idea of the knowledge needed to play this game.

Research

Coding game

Creating interesting game background to help leaners learn coding from scratch
Competitions similar to OJ but multi-player
Different types of games to improve coding skills
Lacks knowledge of some advanced algorithms

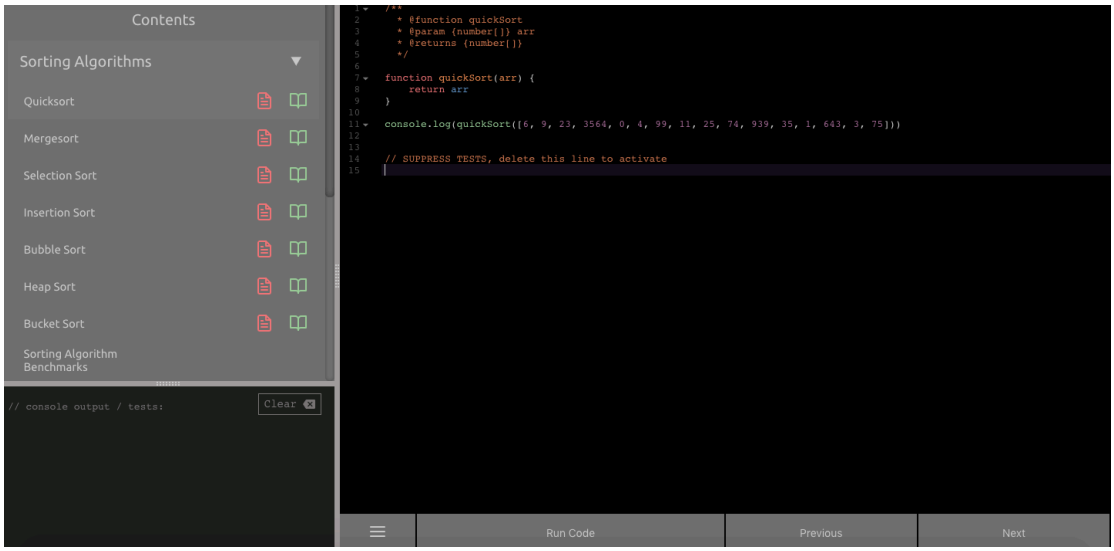
Fancy visual effects are used as an illustration of some rather abstract ideas in programming which makes this platform quite fascinating for new coders who want to make a start. However, most of the content is for beginners rather than experienced programmers. They also offer multi-player competitions where one can challenge friends, schoolmates or coworkers, which is fun and engaging, but there is less visualisation in some of the higher level problems. In addition, those coding challenges are less difficult than those on lots of OJ platforms.
https://www.codingame.com/start



CS-Playground-React

A simple in-browser JavaScript sandbox for learning and practicing algorithms and data structures, lots of classic sorting algorithms as well as data structures that are frequently used are included in this platform. Some challenges are provided at the end to test one's knowledge and make it more fun. In addition, it also offers solutions when one gets stuck, and comes full of links to helpful articles, tutorials, and other resources but that is provided as an external link which makes it less interactive. In conclusion, this website does teach you some very important data structures and algorithms but not in a game context.

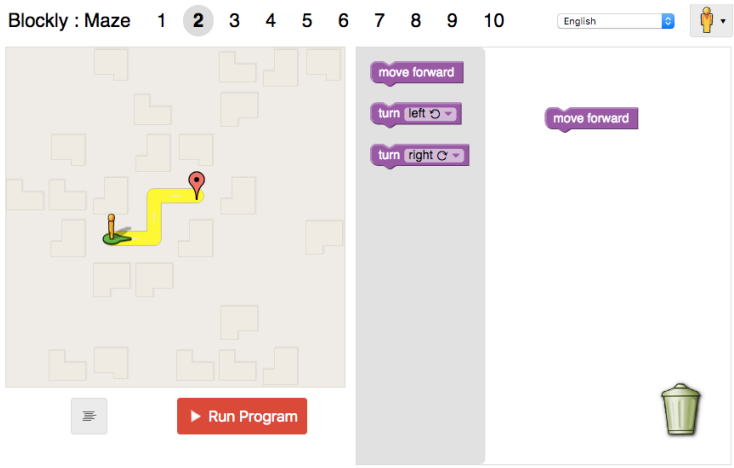
http://cs-playground-react.surge.sh/



Teaching Children Computer Programming

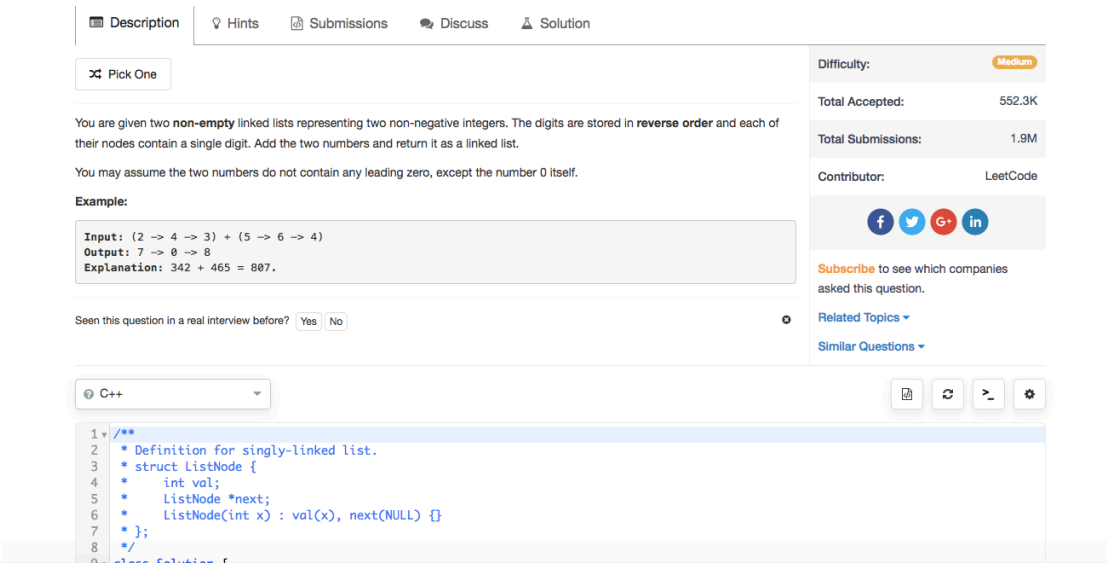
There are lots of introductory games for children as a taste of programming, for example, the Scratch programming language. And this website also offers 6 free games for teaching kinds programming. Although these games are very interesting and easy to play, they are designed to be a very brief introduction to programming which is not what I intended to do.

https://educators.brainpop.com/2014/09/26/6-free-games-teaching-computer-programming-kids/



Online Judging system

https://uva.onlinejudge.org/
https://leetcode.com/
http://codeforces.com/

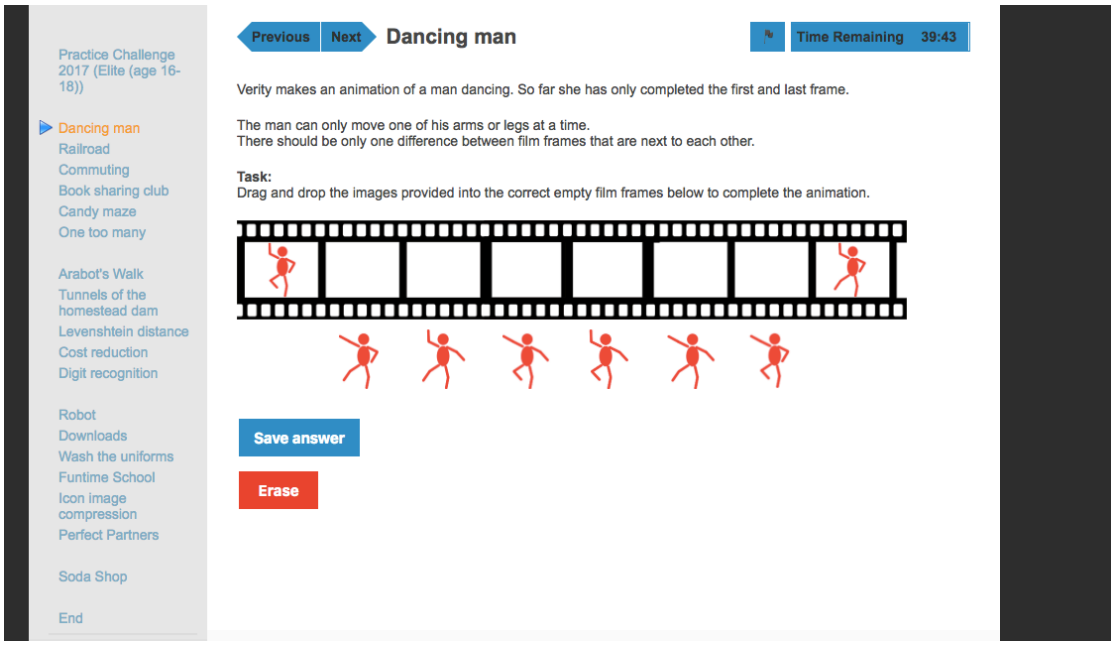


These are all very well-known online judge systems where programmers can practice solving algorithmic problems and submit their solutions to judging systems. Feedback can normally be given in a few seconds. Some competitions may also be held regularly on these platforms. They are all perfect places to improve coding skills, but those problems are all designed by experts and tend to be very challenging. And it is often the case that there is no visualisation of problems available on these websites, which makes solving these problems even harder.

UK Bebras challenge

This challenge is held annually and introduces computational thinking to students in different age groups. The main part of this challenge is to solve puzzles that require logical thinking rather than prior knowledge in computer science. Web-based human interaction is also available so that the participant can interact with the computer to obtain the solution.

http://www.bebras.uk/students.html



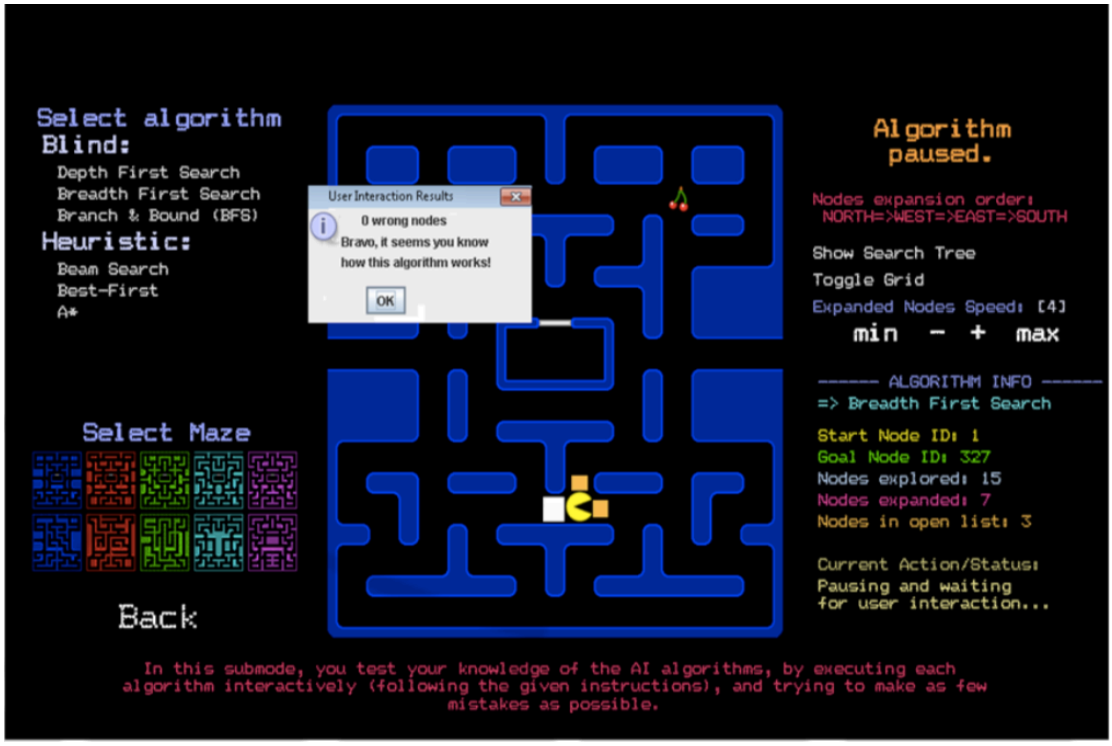
Ticket to Ride – a board game

Ticket to Ride is a cross-country train adventure where players collect cards of various types of train carriages that enable them to claim railway routes connecting cities in various countries around the world.

Students connect one city to another through the missions that they choose. They will come across the implementation of some path finding and minimum spanning tree algorithms, such as Kruskal's, Prim's and Dijkstra's algorithm.
https://www.daysofwonder.com/tickettoride/en/

An Educational Game for Teaching Search Algorithms

This is an article about using a Pacman game designed by the university team to teach searching algorithms such as DFS, A*, etc. This game consists of detailed explanations and visualisations of these algorithms in a game context. Students who play this game will also have the chance to apply their knowledge in this game in order to solve some of the challenges in this game.



Brief conclusion

Having done the research, I realised that there are some similar platforms and projects that are committed to the visualisation of algorithmic problems, though with various approaches and levels. Some coding games have fascinating game context for new coders and makes it much more appealing to beginners. On the other hand, online judging systems provide much more challenging and well-designed problems and feedback on users' solutions after they submit their codes but most of the time images are not available. If we go back to my initial thought, I intended to combine interesting stories and abstract computational ideas together to make the learning process less painful. This idea is similar to the game mentioned in the article An Educational Game for Teaching Search Algorithms. The game comes with explanations of great details and vivid illustration. Human interaction is achieved by asking students to apply what they learned in another mode of the game. This is the closest solution to my problem and thoughts, but there is still something to add.

Essential features

In the project mentioned above, only searching algorithms are included and they are all demonstrated in the same game (which does have some advantages). I want to add more algorithms of different categories and different games might be used to introduce different algorithms. It will consist of three features:

1. The game is a combination of OJ (online judging) and the game "Pacman for teaching search algorithms", two of the existing solutions to my problem. Therefore, most of my games will be based on problems from online judging systems (might be abstracted or decomposed). In other words, OJ problems will be visualised in my game. This is due to the fact that lots of problems from OJ are described as games or puzzles themselves, which already adds fun to my game and makes it easier to adapt them. In addition, qualities of questions from OJ are generally very good and I will also try my best to select the most classic ones to ensure leaners can have a good learning experience.
2. There are a range of games included in my project and most of them will have different levels of difficulty. Brute force may be able to address some of the easiest levels but will become too inefficient to be used while coping with harder levels. This design is based on the idea that efficiency is usually the key part of an algorithm design when the data scale becomes very large, which is often the case in the real world.
3. Last but not least, this game will be able to run offline. Bizarrely, nearly all existing solutions require users to have a stable Internet connection. Although publishing a game online will make it accessible to more people, being able to run offline gives users the opportunity to access it anywhere and anytime they like. And the game could run more smoothly on a local computer.

Limitations

There are two main limitations to my solution. The first one is that the formal theories behind each game are not included in this game and users will have to find out themselves exactly how a problem is solved by a computational method and how this method is developed step by step. The reason why theories are not included is that there are already lots of excellent books or websites available, designed by expert computer scientists, and they tend to be very good resources for learning theories. The main focus of this game is to provide visualisation which can assist learners in their learning process and it is not meant to be a replacement for further study of those theories. This game is therefore not suitable to be used as a full learning guide.

The second limitation is that the visual effect of this game will not be as good as that on some of other platforms which appeared in the research. Although effort will be

made to demonstrate the functionality of algorithms as much as possible, fancy visual effect is not the main focus of this game. Some of the games may have already done the abstraction and may be presented to the user differently from the original description. This could potentially make the game less interesting to play for some of the users.

Requirements

No.	Description	Comments	Priority
Design			
1.	Clear instruction on how to play the game	So that a user can easily start to crack on with the game	Must
2.	Player movement	The player is able to move	Must
3.	All information required is displayed	So that the user is able to play the game	Must
4.	Game running correctly and logically	e.g. illegal moves are not allowed	Must
5.	Problem illustration as straightforward as possible	Many of the games in my project will be a simplified version of some rather complicated problems, so it is important to give clear illustration	Must
6.	Having seen a similar problem is helpful but not essential to be able to play	A game level might not contain detailed explanation or tutorials such as examples on how to operate, but operations will be designed so that most users can figure them out intuitively	Should
7.	Extra guidance provided if a user does not understand the problem	If a user is having difficulty on how to play the game or understanding the problem, there will be	Could

		extra help available	
8.	Challenges are offered in various levels of difficulty	So that users can build up knowledge and understanding of an algorithm	Should
9.	Some illustrations of characters are included	To add more fun to the learning process	Could
User experience			
10.	Game can run smoothly and provides good user experience	Smooth running can give the user the willingness to keep playing	Should
11.	Program does not contain serious bugs	So that the game will not crash unexpectedly in most situations	Should
12.	Essential functions are implemented and available to the user	e.g. the user does not need to start the game again if he/she took a wrong step as undo button is provided	Must
Educational aspect			
13.	The user needs to think strategically in order to pass the game	As the primary purpose of this game to help user learn knowledge	Must
14.	The game is as interactive as possible	So that user will not give up the game easily if he/she is stuck	Could
15.	This game can encourage people to learn more about these algorithms rather than deter them	Another main focus of this project is to make the learning process of some complex algorithms more enjoyable, therefore the game is designed to motivate learners	Should
16.	Solutions are straightforward for	Although explanations on how a solution is	Should

	players to follow	achieved will not be included, the correct answer will be displayed and made easy for a user to follow	
17.	Principles behind games are logically strict	Since this game has educational purposes for advanced intermediate and advanced learners, it is vital to ensure the principles are accurate and logical	Must
18.	Stories are interesting to most players	Again, this can make a user enjoy more about the learning process	Could
19.	Puzzles are chosen based on how classic and how often they are used to solve problems	There is not much point for a programmer who is not very experienced to learn algorithms that are less often used	Must
Code itself			
20.	Idea of OOP is used properly throughout	There are lots of ideas in object-orientated programming, e.g. class, inheritance, private, etc. These ideas should be used accurately and correctly in the code.	Must
21.	Code is as elegant as possible	So that it would be easier for others and to read	Could
22.	Code is nicely commented	To make review process easier and more efficient	Could
Stakeholders' requirement			
23.	Puzzles are carefully selected so that they	By basic, they refer to algorithms such as	Must

	are neither too obscure nor too basic.	binary search, while the network flow would be considered too advanced	
24.	Puzzles are designed to deepen the understanding of algorithms rather than broaden the knowledge		Must
25.	Puzzles are designed to as interesting as possible with some illustrations available	For example, some pictures could be used	Could
Others			
26.	Hardware	A computer that is able to run this game would be sufficient. As this game does not take many CPU resources, nearly all computers nowadays will be able to run this game.	
27.	Software	Windows, macOS or Linux installed. These three operating systems support Python which is what this game is written, therefore it would be essential for a device to have one of these operating systems installed	
28.	Python interpreter	This is used to run the source code of this game.	
29.	Pygame library	This library provides lots of functionality that enable a programmer to write a game. My game will be mainly based on this	

		library.	

Success criteria

The success criteria are indicated with a “must” as their priorities in the requirements.

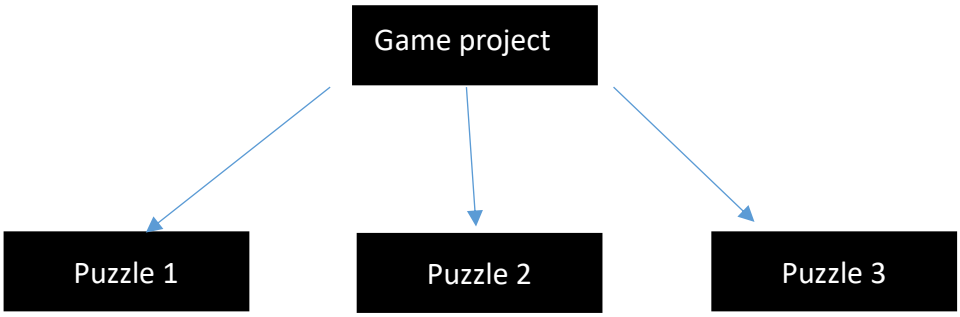
Design

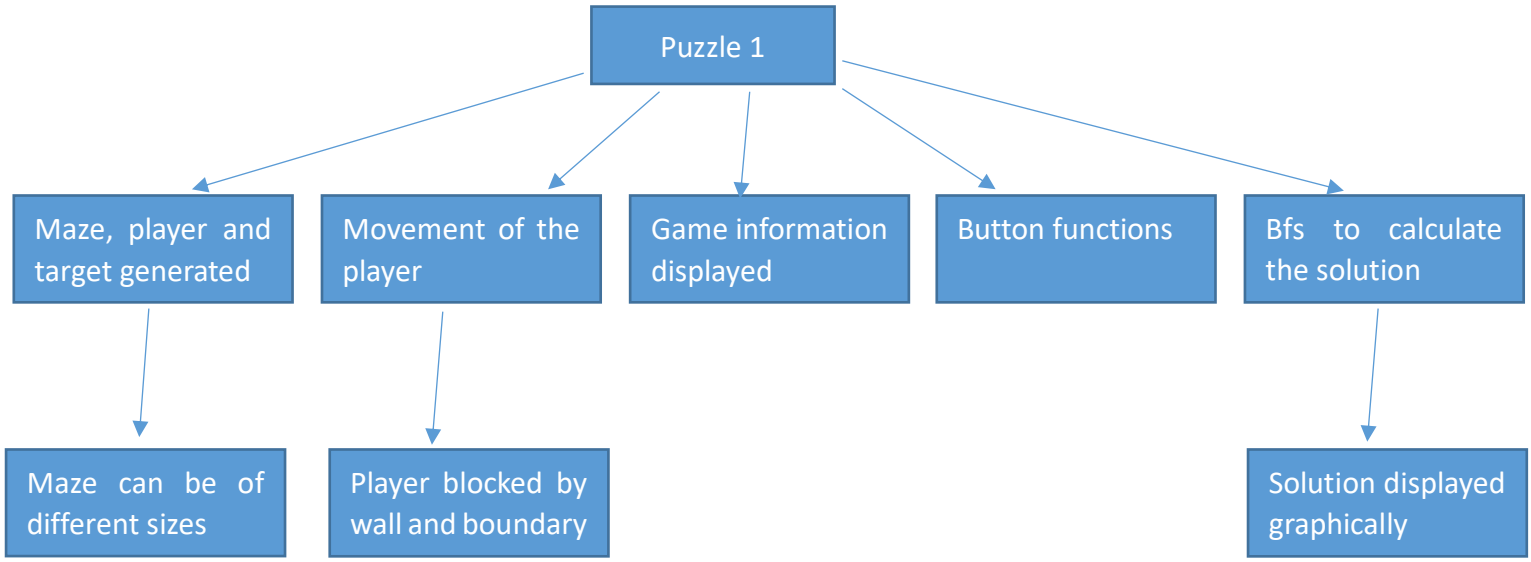
The game is divided into three different puzzles: breadth-first search, shortest path and knapsack problem. Each of which is solved by a classic algorithm. These games are chosen based on my own programming experience. From the stakeholders' point of view, they would also like the most classic challenges to start with rather than go directly into more advanced ones.

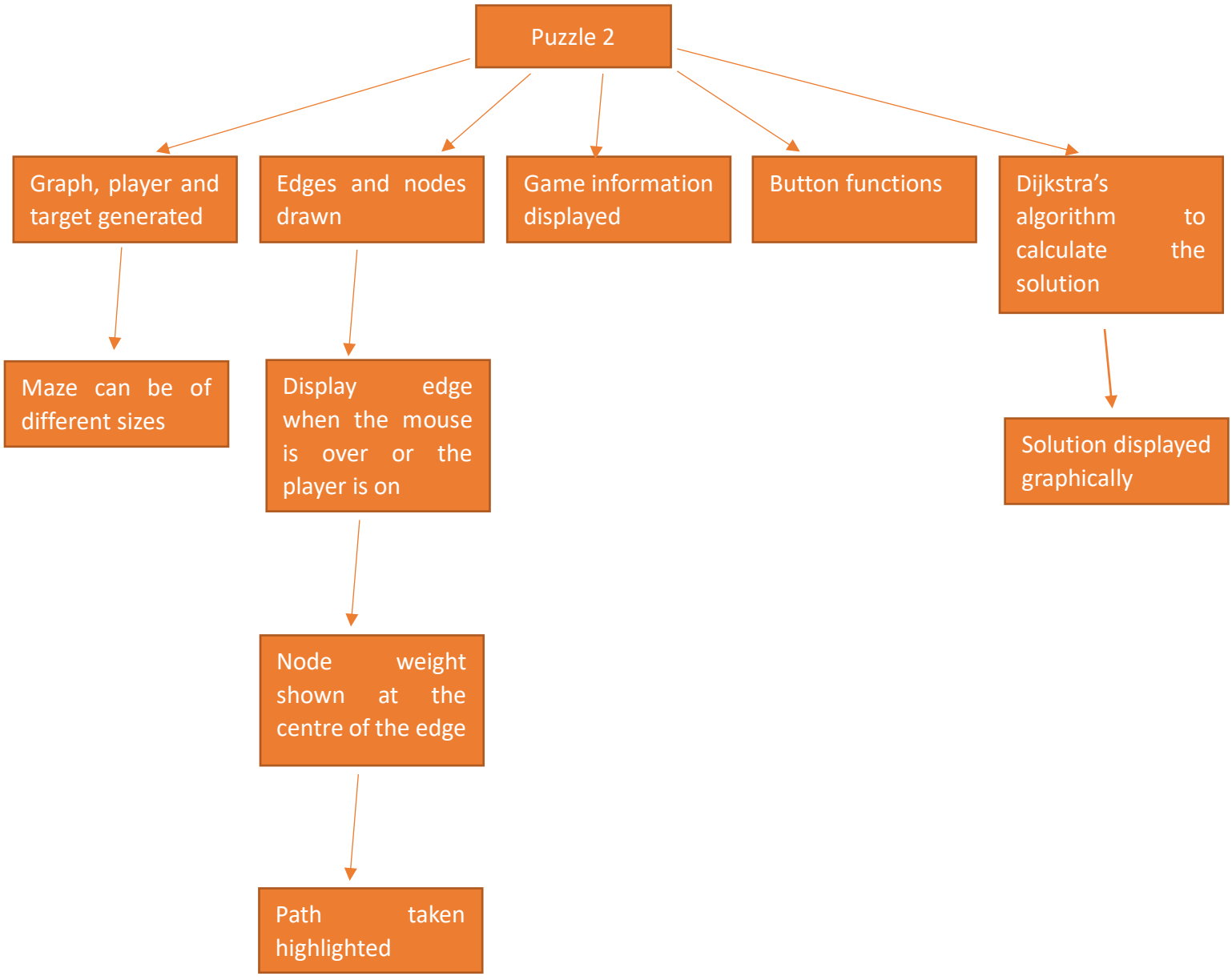
Some of these puzzles will be available in different levels of difficulty so that this can make the player apply knowledge to slightly unfamiliar situations and build up their understanding of the algorithms involved. It is also stated in the stakeholders' requirement that the game is meant to deepen the understanding of algorithms.

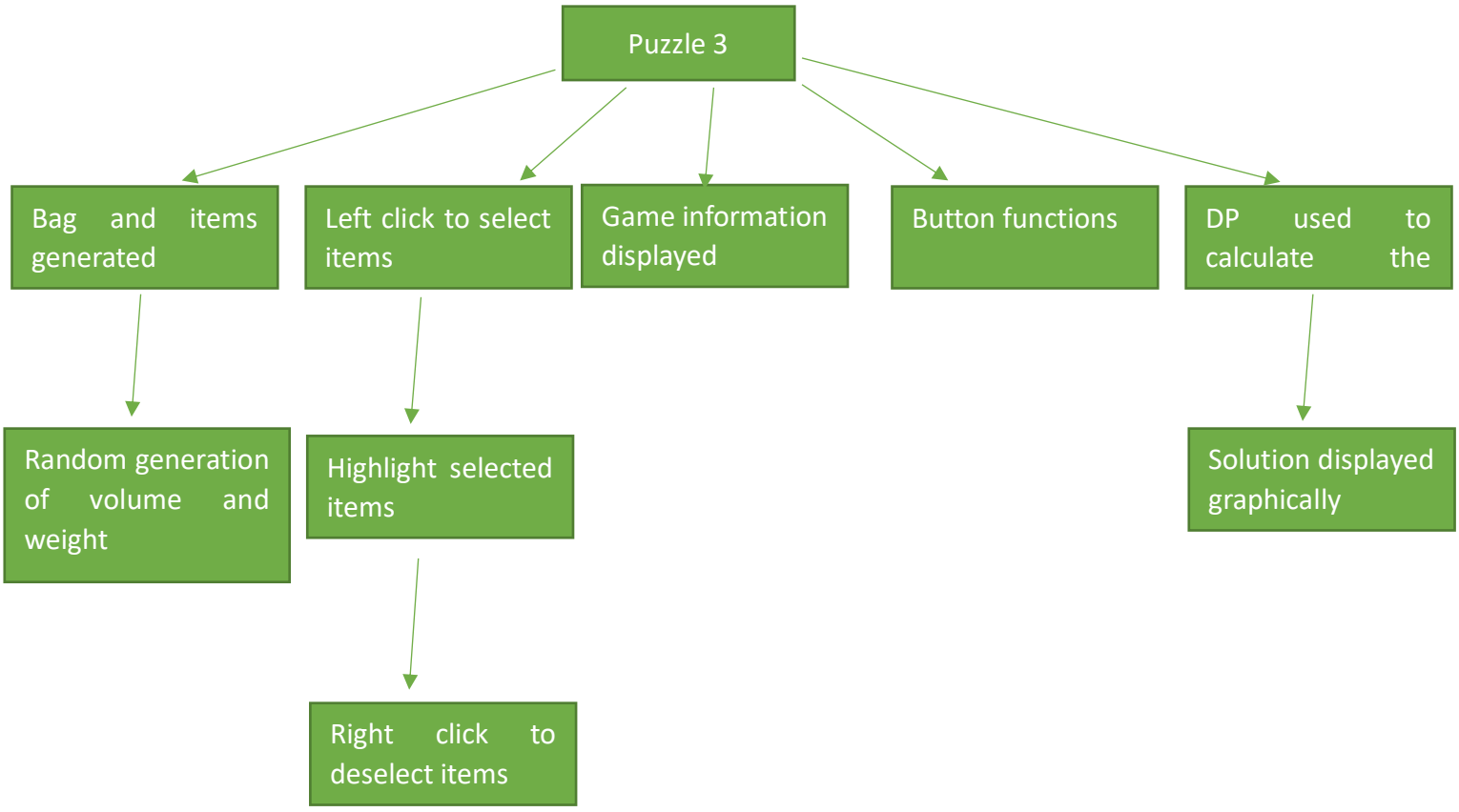
Top-Down Design

This is the overall structure of the game when broken down into smaller problems, and it also contains the order in which smaller problems are to be solved. More detailed description will be in the interface design.



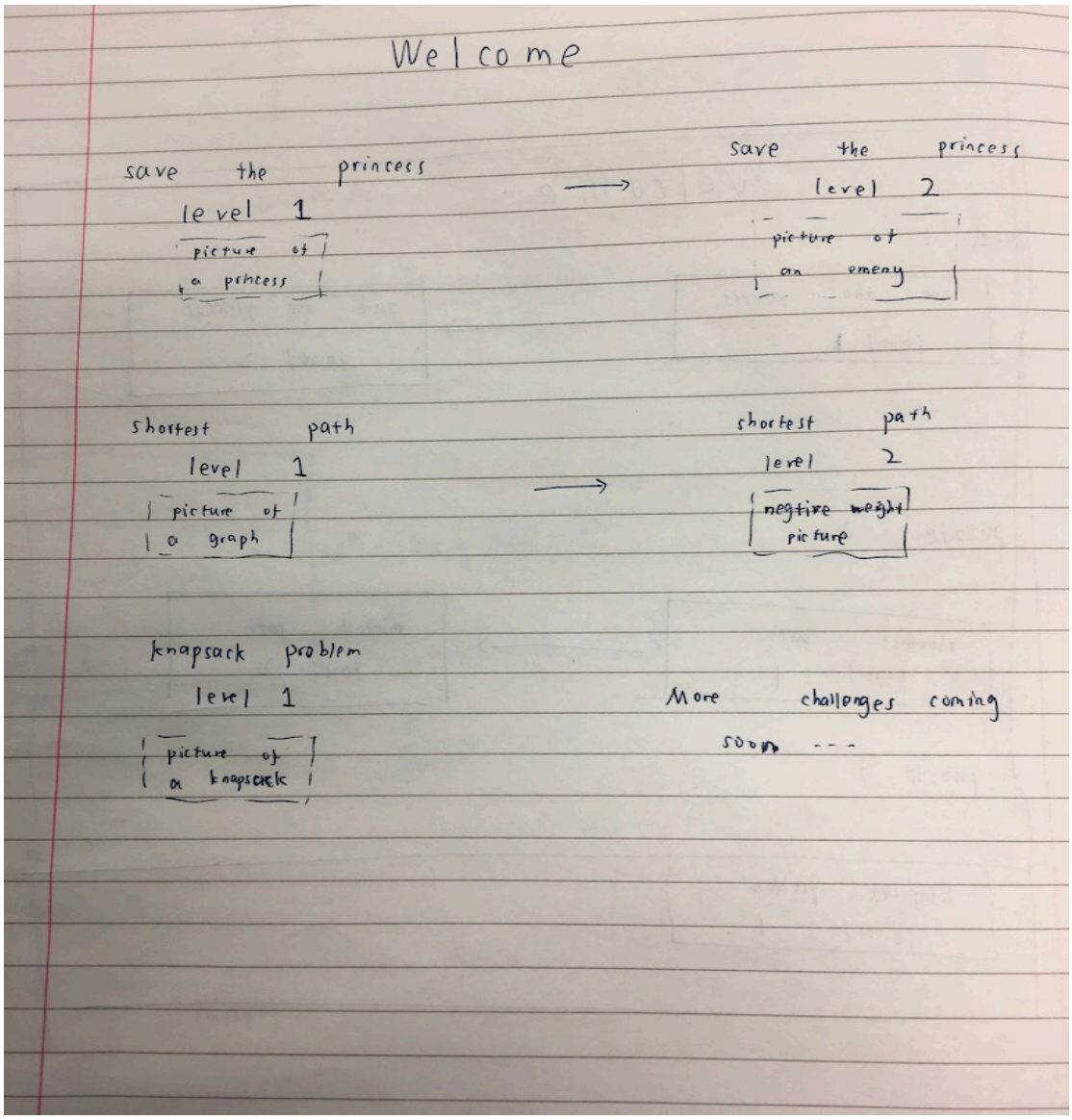

Puzzle 1
Maze, player and target generated
Movement of the player
Game information displayed
Button functions
Bfs to calculate the solution
Maze can be of different sizes
Player blocked by wall and boundary
Solution displayed graphically


Puzzle 2
Graph, player and target generated
Edges and nodes drawn
Game information displayed
Button functions
Dijkstra's algorithm to calculate the solution
Maze can be of different sizes
Display edge when the mouse is over or the player is on
Solution displayed graphically
Node weight shown at the centre of the edge
Path taken highlighted


Puzzle 3
Bag and items generated
Left click to select items
Game information displayed
Button functions
DP used to calculate the
Random generation of volume and weight
Highlight selected items
Solution displayed graphically
Right click to deselect items

Interface design

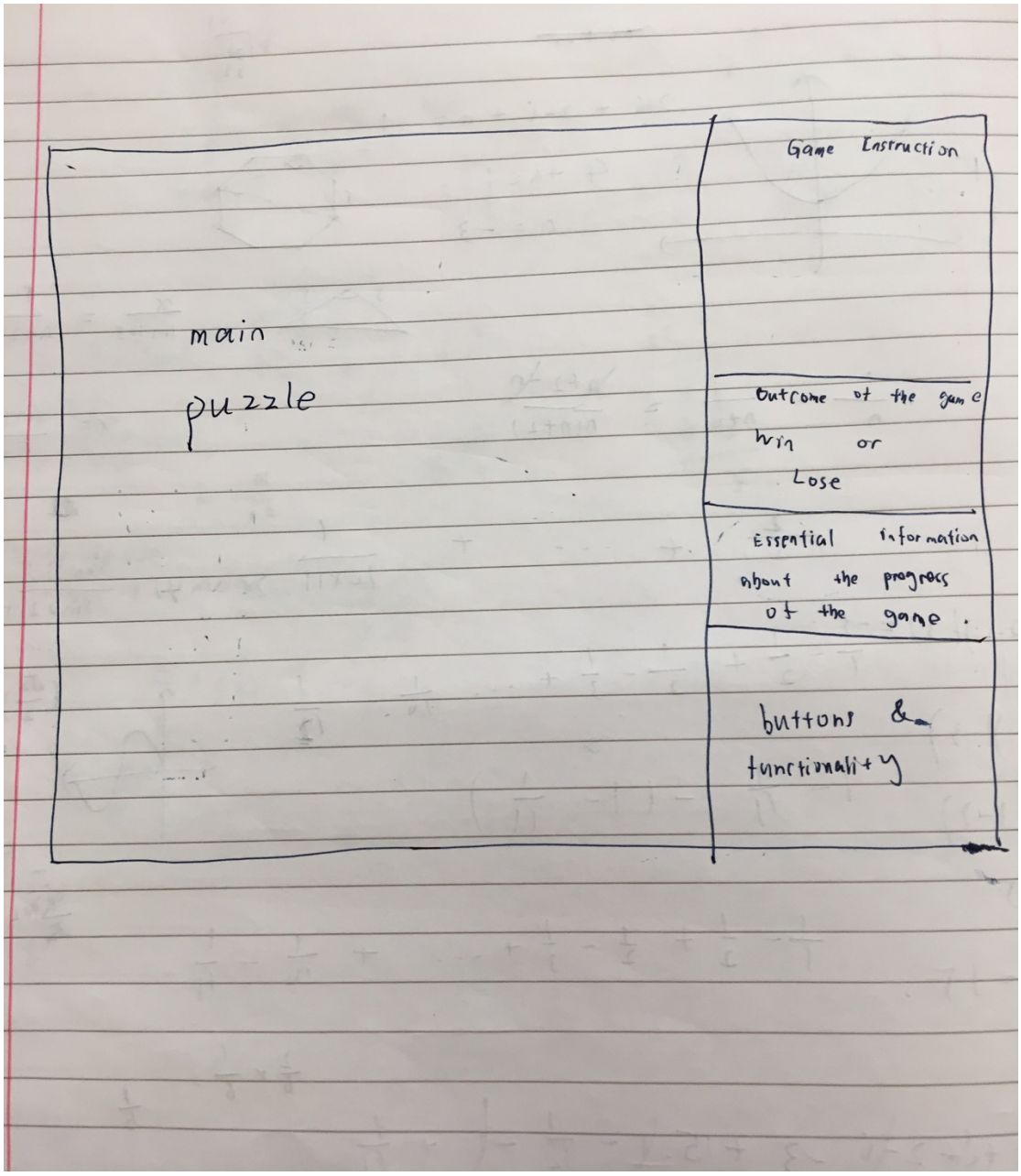
Puzzle Selection



In the level selection interface design, different levels of the same challenge are offered to the user so that they can build up their understanding of the game step by step and it meets the requirement: “Challenges are offered in various levels of difficulty”.

Below every level that the user can select, there will also be a picture describing the feature of that particular level. This is included so that the user can easily see the main characteristics of the level and they can come back if they wish to have another go at a particular challenge. It also makes sure that the game has a straightforward and vivid illustration, which is required in the success criteria and adds fun to the game itself.

General layout of a puzzle

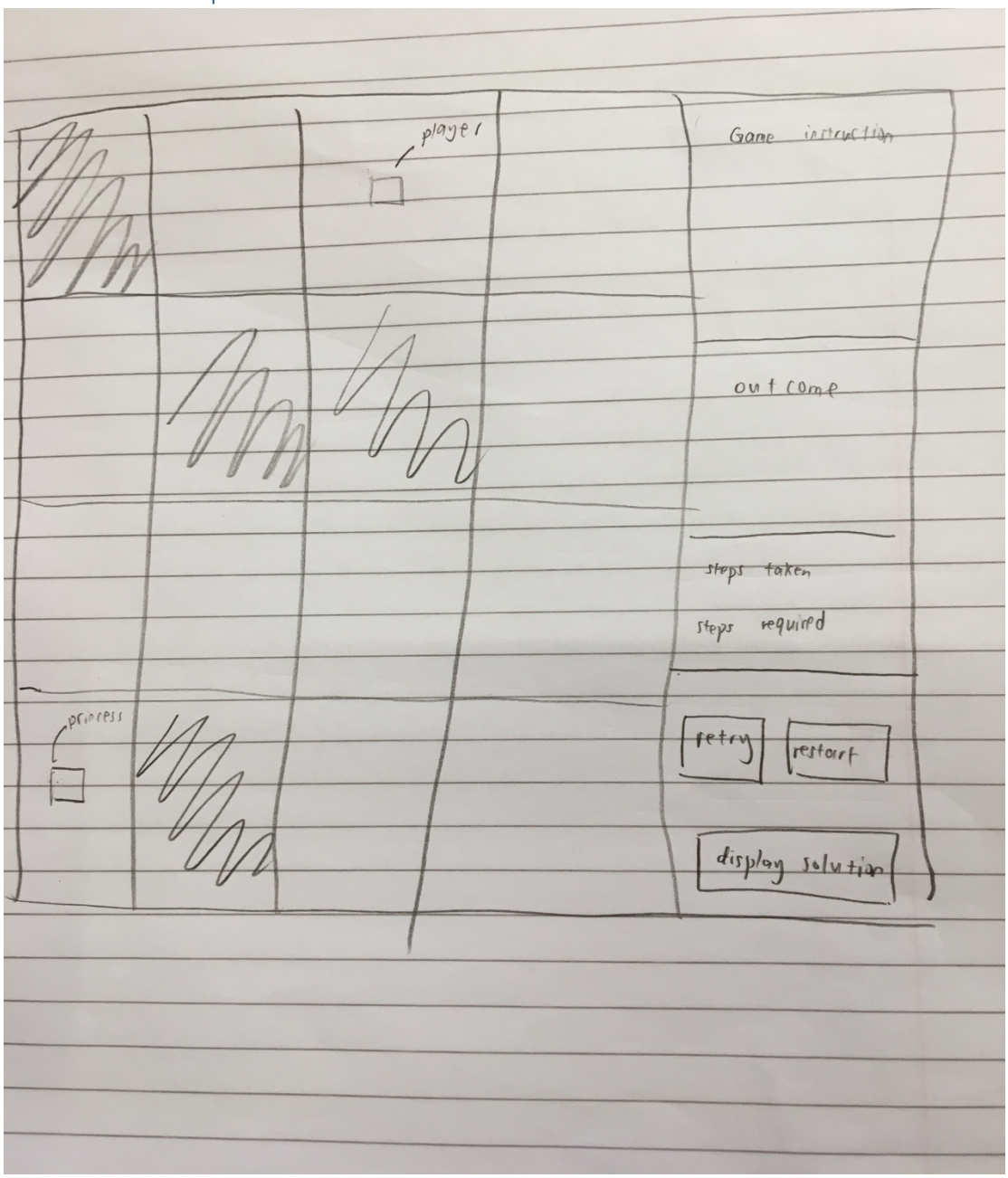


Generally, the screen is divided into two sections: the main puzzle, and the game information. The main puzzle section will obviously display the progress of the game graphically while the user is playing the game. And the information section gives essential information about the game which has four parts:

- Game instruction: a brief introduction to the user on the objective of the game and how one could play the game;
- Outcome of the game: it shows whether the user has successfully accomplished the objective or there is a better solution to the game;
- Current progress of the game and the optimum solution: this will depend on the nature of the puzzle;
- Buttons: used to provide essential functionalities to the user. All puzzles will

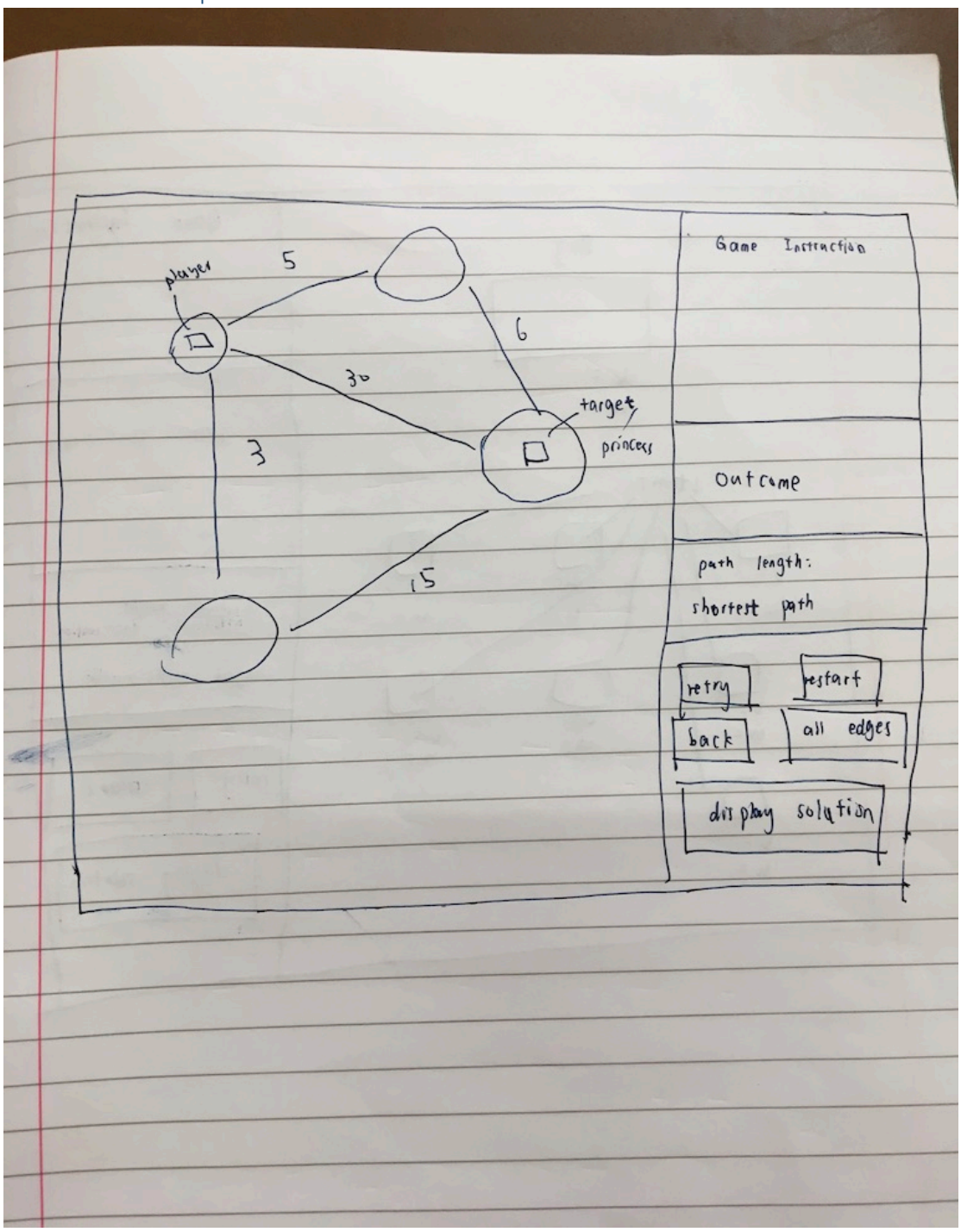
have three functions in common:
Retry – start the current puzzle again;
Restart – restart the current puzzle with different difficulties;
Display solution – solution of the puzzle will be displayed if this button is pressed.

Puzzle1: Save the princess



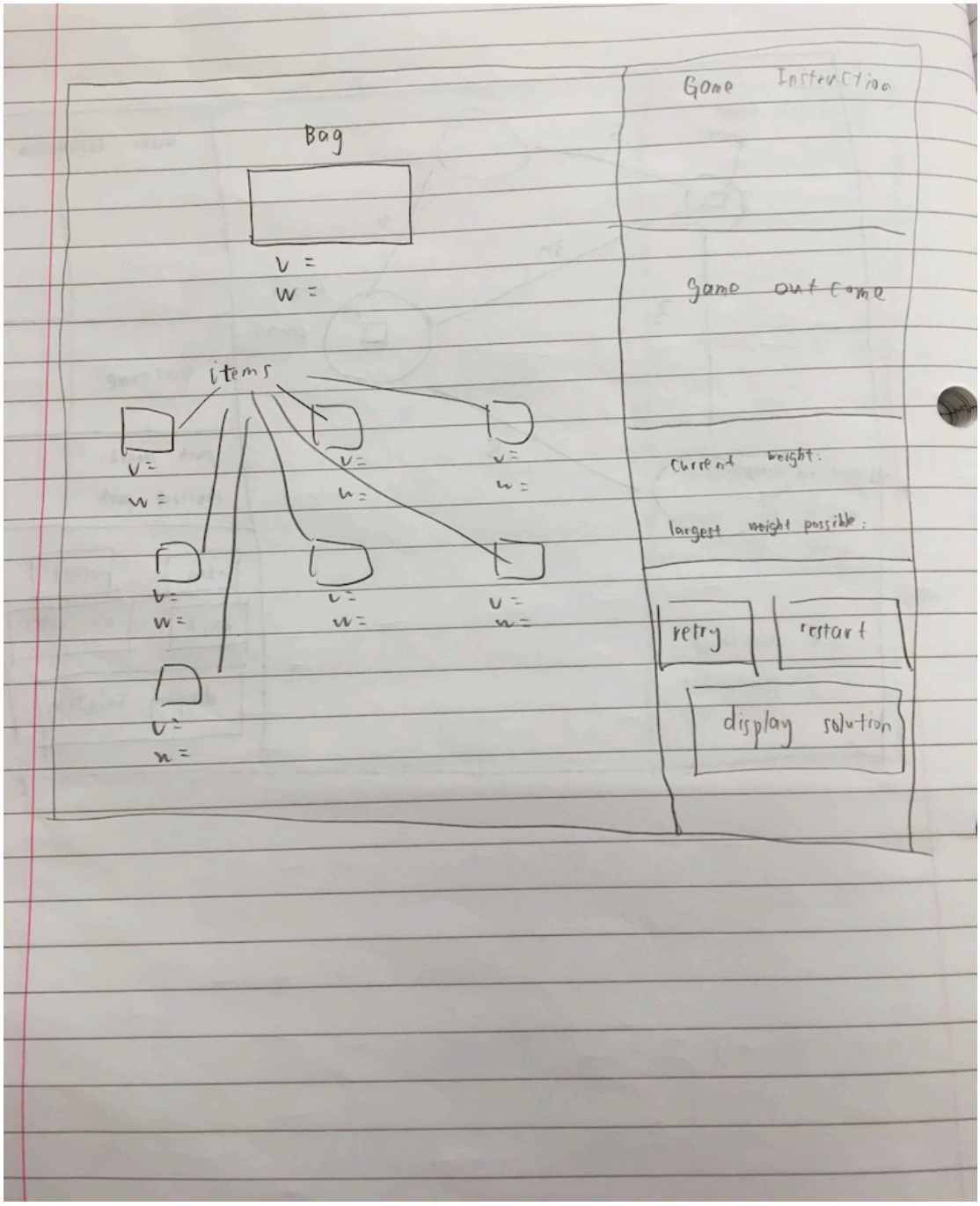
This is the “save the princess” challenge level 1, which is a classic application of breadth-first search algorithm. The user needs to move the player to reach the princess in the minimum number of steps. The maze consists of a number of cells where shaded ones are not reachable. The maze will be generated on a random basis and the size of it will change each time. This is to test the player’s understanding of the algorithm as opposed to memorise the solution. The player has to move across the maze and pass only those white squares. That is why, in the top down design, the player movement is included. On the right-hand side, basic information such as how many steps the user has taken and what the minimum number of steps one could take to reach is shown to the user, so that one can have an idea of how close he/she is to the final solution. This puzzle has all three basic functions of all puzzles.

Puzzle2: Shortest path



This is the second puzzle that requires the player to move from a starting node to the destination node with the shortest path on a graph. Again, the graph is randomly generated due to the same reason as in puzzle 1. It is also important the graph is displayed clearly and hence this is ensured by breaking it down into four smaller problems in the top down design. This puzzle is adapted from another classic shortest algorithm – Dijkstra's algorithm. Some specific information of this game includes how long the path is that the user has taken and what the optimum solution is.

Puzzle3: Knapsack problem



Again, the knapsack problem is a well-known mathematical problem and also a perfect example for the programming technique – dynamic programming. In this challenge, the player needs to select items that they think will give the maximum overall weight within a limited volume. And apart from the basic puzzle generation, this puzzle is different from the previous two in that it involves selecting and deselecting and hence it the selection is broken down into three steps.

Usability

In terms of the usability of this game, various designs can be found to make the game more usable to a user. In terms of general elements that all puzzles have, the following can be considered to make the game more usable:
Game instructions designed to help a user get a basic understanding of the game;
Information can be found that is dynamically changing as the game progresses to help the user make decisions about what to do next;
Outcome of the game will also be displayed once the target has been reached and it can also give further advice on whether there is a better solution to be found;
Buttons are designed to give the user better experience while he/she is playing the game. For example, a user might want to restart the game if he/she made some mistakes or simply wants to try a new challenge. Graphical solutions are also available through the solution button.

There are also specific designs in each puzzle to ensure usability:
Puzzle1: Arrow keys to move the player in a maze, which would be quite straightforward for one to do when seeing something in a maze;
Puzzle2: Clicking the node to move the player can be quite an easy operation to recognise and it is also included in the game instruction. In addition to three basic functions that all games have, this puzzle has two extra ones—the back button and the all edges button. The back button allows the user to undo one movement, which is naturally what someone wants to do if he/she makes a mistake and this is also one of the essential functions required in the success criteria. The all edges button will display all edges when it is pressed, which can give the user a general idea of what the whole graph is like;
Puzzle3: The user is able not only to select items they want to put into the bag, but also deselect by right clicking the item, which can make it much easier if the player changes his/her mind during the game. Highlighting selected items can help the user see at a glance what choices he/she has made and what is the next thing to choose.

The game information section is designed to ensure the usability of the game while also reflect the following requirements:

- Clear instruction on how to play the game, which is achieved by giving game instructions and showing current progress to the user;
- Having seen a similar problem is helpful but not essential to be able to play, as operations required to play the game are all very intuitive;
- Essential functions are implemented and available to the user.

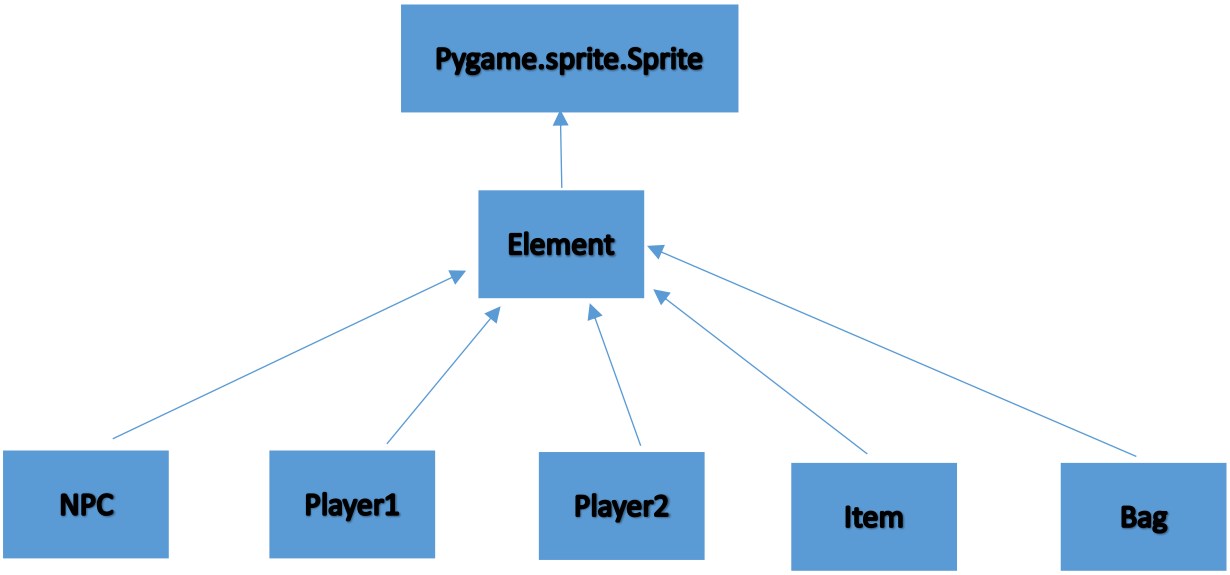
Classes, key variables and data structures

Inheritance diagram

Now the different interface designs of the game are ready, and it is time to go on to the code design. Based on the nature of all the elements in three puzzles, they will all inherit from the built-in Pygame class Sprite.

There are three reasons why these relationships are designed like this:

1. All objects in different classes are relatively independent of each other. For example, a player1 object will not interact with player2 at all, and the player1 is obviously not player2. Therefore, these objects do not inherit from each other according to the “is a” rule.
2. However, there are lots of similarities between these objects. For instance, the item and bag objects share the position and colour attributes. Because of this, they all inherit from the super class Element. This saves time writing the same code and makes it easier to manage these objects.
3. Finally, all of these classes are also child classes of the Pygame built-in class sprites. This ensures that some of the attributes, such as rect, do not need to be implemented again and it is easier to draw these objects on the screen.



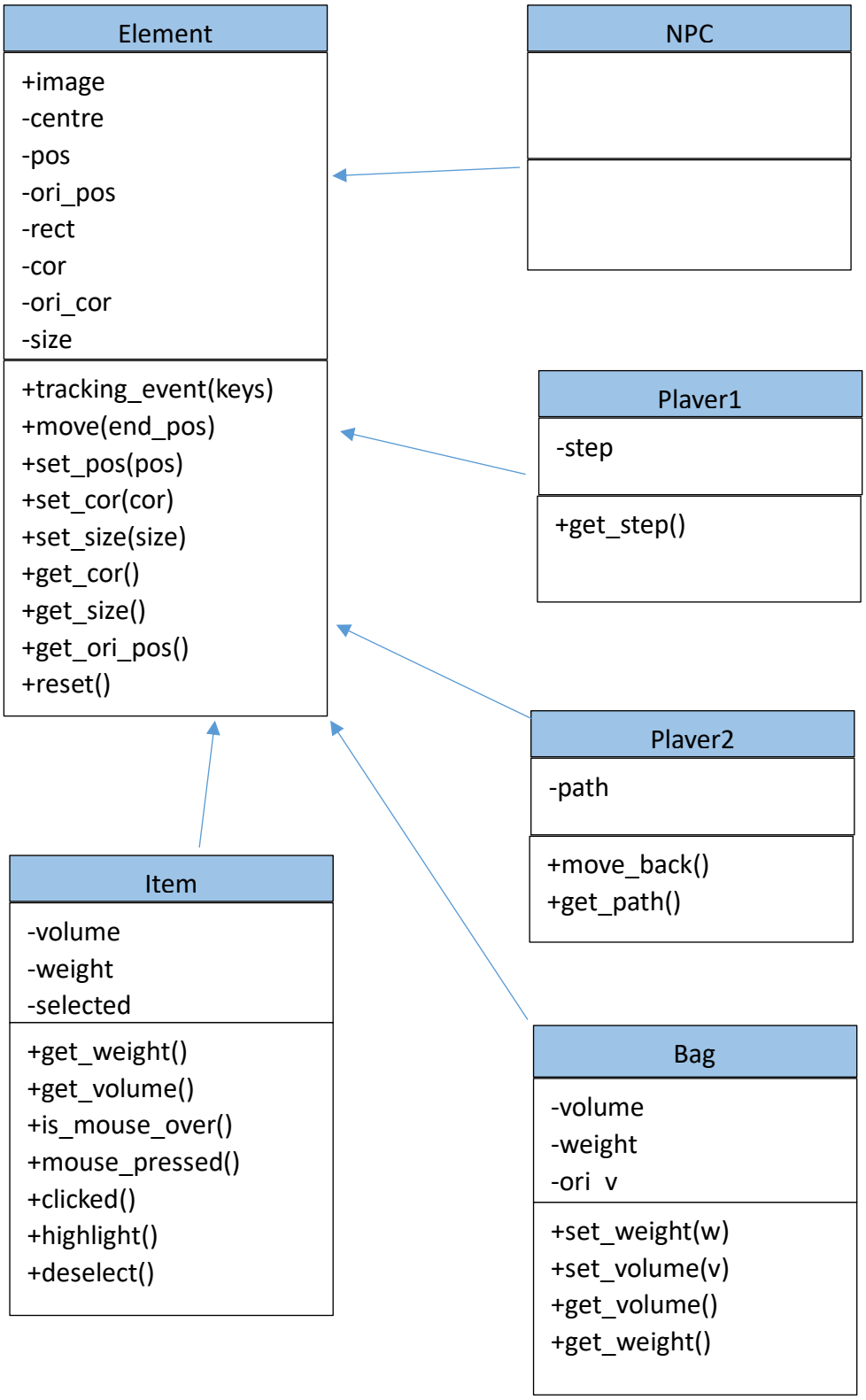
Class diagram

The class diagram of different classes are as follows. In each rectangle, a plus denotes that it is a public attribute or method whereas a minus means it is private.

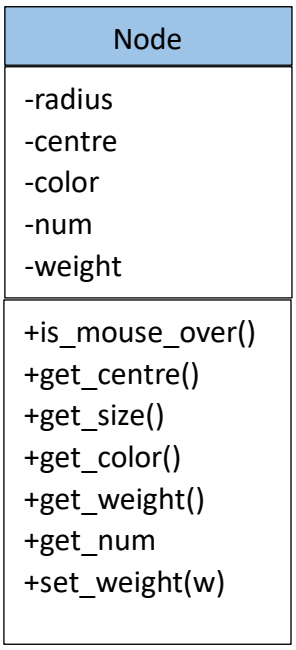
Here the Element class is the super class of all those classes that inherit from it, such as Player1 and item. The NPC class, as its name suggests, is a template of non-player characters such as the princess in the first puzzle. Player1 and Player2 represents players in the first and second puzzle respectively. Finally, the Bag and Item class are used specifically in the third puzzle to create item and bag objects.

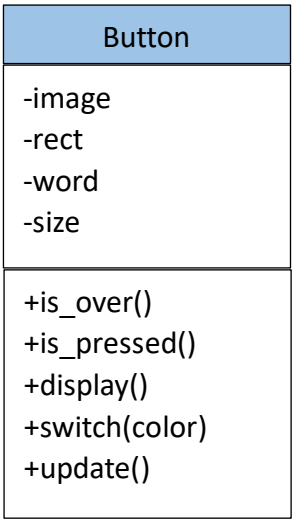
In terms of the attributes and behaviours these objects, apart from the basic “getter” and “setter” methods, there are:
tracking_event(), which is used to check if there is an input from the user. This could be from the keyboard or the mouse, which is essential as the game character needs to respond to the user input;
move(), called when an object needs to make a move;
reset() will reset all attributes to the initial value. This is needed when the user wants to restart the game;
move_back() will undo the last move when the user made a mistake;
clicked() checks whether an item is clicked or not;
highlight() will be called when an item is selected and highlighted;
deselect() is needed when a user wants to deselect an item.

Explanations on how attributes are used in each of these classes will be included in the key variables.


Element
+image
-centre
-pos
-ori_pos
-rect
-cor
-ori_cor
-size
+tracking_event(keys)
+move(end_pos)
+set_pos(pos)
+set_cor(cor)
+set_size(size)
+get_cor()
+get_size()
+get_ori_pos()
+reset()
NPC
Plaver1
-step
+get_step()
Plaver2
-path
+move_back()
+get_path()
Item
-volume
-weight
-selected
+get_weight()
+get_volume()
+is_mouse_over()
+mouse_pressed()
+clicked()
+highlight()
+deselect()
Bag
-volume
-weight
-ori v
+set_weight(w)
+set_volume(v)
+get_volume()
+get_weight()

These are the standalone classes that are unique to each of the puzzles. For example, the Tile class is used in the first puzzle to create tile objects in a maze. Node objects in the second puzzle are instantiated based on the Node class. It is necessary as it is an essential element in a graph used in puzzle2. Last but not least, the Button is used as a blueprint for all buttons used across all three puzzles.





Key variables

Variable name	What it is used for	Variable scope
centre	The central coordinate of an object	An attribute
pos	The top left coordinate of an object	An attribute
cor	The cardinal number, e.g. (1,3)	An attribute
size	The size of an object	An attribute
step	Number of steps in puzzle 1	An attribute
volume	Volume of an object in puzzle 3	An attribute
weight	Weight of an object in puzzle 3	An attribute
rect	Pygame objects used to display an object on the screen	An attribute
color	Colour of an object	An attribute
done	Control the main program loop	Global variable
curPuzzle	Determines the current puzzle	Global variable
screen	Pygame object for the screen	Global variable
INF	Infinity	Global variable
font	Font of letters to be displayed	Global or local variable

Data structures

Name	How it is used
Sprites groups	It is used as a global object that will be updated in the main program loop. Polymorphism is achieved by calling the update method of each of the object (may be of different classes) in the group
A queue	It will be used to implement breadth-first search in the solution of the first puzzle
A graph stored in an adjacency list	It will be used to implement Dijkstra's algorithm in the second puzzle
An 2d array	It is used to store intermediate results of dynamic programming, which is how the third puzzle is solved.
Lists	Used across three puzzles to store, for example, all items to be selected in the third puzzle.

Pseudocode algorithms

Now that the overall structure and function of each class has been defined and the relationship between different classes has been specified, more specific algorithms and pseudocodes are needed to describe the solution in more detail.

As there are lots of similarities between different puzzles, only algorithms specific to a puzzle will be demonstrated in the pseudocode.

A general design of puzzles

Before going on to the specific features of each puzzle, a general design of the UI will be needed as described in the general layout of a puzzle. This includes game instructions, game progress, buttons, etc.

Game instructions will be displayed using a procedure called display_info().

```
procedure display_info():
    // game instruction
    game_instruction = []
    game_instruction.append(font.render("Game Instruction:", True, RED))
    // some more instructions specific to a puzzle will be added here


    screen.display(game_instruction)

    // game information
    // the outcome of the game will be added here
    // game progress specific to a puzzle

endprocedure
```

There are two buttons shared by all puzzles, restart and retry, subroutines that implement the functions of these two buttons can also be designed prior to more specific design of each puzzle.

```
procedure retry()    // reset the player position and the maze stays unchan
ged
    if retry_button.is_pressed():
        player_position.reset()
    endif

endprocedure

procedure restart()  // the maze is re-generated
    if restart_button.is_pressed()
        all_sprites_group.empty()
        game.reset()
        self.pre_update()
    endif

```

Requirements met in this design

Requirements	Designed in which subroutine	How it is met
Clear instruction on how to play the game	display_info()	Game instructions will be shown via this procedure
All information required is displayed	display_info()	Game progress, which is important to the user, will be displayed
Essential functions are implemented and available to the user	retry() and restart()	These two are the most basic and essential functions for each puzzle
The game is as interactive as possible	display_info()	The game progress will be dynamically updated in this procedure

Puzzle 1: save the princess

The initialise procedure allows the maze of puzzle 1, as well as the player, and the princess to be generated ready to be displayed. It is also possible to use this method to re-generate these elements for a new challenge. If this procedure was integrated into a class, it should not be the constructor of the class, and the reason for that is that a constructor will only be called when an object is constructed, whereas the puzzle needs to be generated multiple times and with a different grid size each time, therefore writing a separate subroutine that can generate the maze randomly each time will be more suitable.

```
procedure initialise()
    # 1=wall 2=player 3=princess
    maze_num = random(5, 30)
    maze = [maze_num, maze, nun]
    tile_list = []

    nSpecialElement = random(0, maze_num ^ 2 * 0.5)
    # randrange [a,b), 60mod of the maze is wall
    playerCor = random(0, maze_num ^ 2)
    maze[playerCor // maze_num][playerCor mod maze_num] = 2

    while True
        princessCor = random(0, maze_num * maze_num)
        if princessCor != playerCor
            break
        endif
    endwhile
    maze[princessCor // maze_num][princessCor mod maze_num] = 3
    nSpecialElement -= 2
```

```
        for i= 0 to nSpecialElement

            while True
                wallCor = random(0, maze_num * maze_num)
                if wallCor != princessCor and wallCor != playerCor
                    break
                endif
            endwhile
            maze[wallCor // maze_num][wallCor mod maze_num] = 1
        next i

        sideLength = 500 / maze_num
        for i = 0 to maze_num
            for j = 0 to maze_num

                t = new Tile((j * sideLength, i * sideLength), (sideLength, s
    ideLength))

                if maze[i][j] == 1
                    t.set_color(0)

                else if maze[i][j] == 2
                    size = [sideLength * 0.4,
                            sideLength * 0.4]  # the size of the player will
    make up two fifths of a tile
                    my_player = new Player1(t.get_centre(), [i, j], size, BLU
    E)  # player is centred
                    all_sprites_group.add(my_player)

                else if maze[i][j] == 3
                    size = [sideLength * 0.4, sideLength * 0.4]
                    princess = new NPC(t.get_centre(), [i, j], size, RED)
                    all_sprites_group.add(princess)
                    princess_cor = [i, j]

                tile_list[i][j] = t
                endif
            next i
    endprocedure
```

The optimum solution is pre-calculated after the maze is generated and before the game starts. This is because it needs to be displayed on the screen once the maze has been generated to give the player a guide as to what needs to be achieved. This is calculated in the get_solution method. In this specific challenge, breadth-first search is used to find the optimum solution. The reason for choosing this searching method this is based on two facts: 1. The graph is unweighted; 2. Optimum solution is required. Although there are other shortest path algorithms that will work on this type of graph, breadth-first search is one of the easiest searching algorithms that can solve this problem in terms of implementation.

```
procedure get_solution()
    solution = -1
    solution_list = []
    q = []
    qHead = 0
    qTail = 0
    # every element in q is a list of three integers s[0] row num, s[1]col
umn number;
    # s[2] number of steps, s[3] father
    visited = [maze_num,maze_num]
    dir = [[0, 1, 0, -1], [1, 0, -1, 0]]
    startPos = my_player.get_cor()
    endPos = princess_cor
    q.append([startPos[0], startPos[1], 0, -1])
    qTail += 1
    while qHead != qTail
        s = q[qHead]
        if [s[0], s[1]] == endPos
            solution = s[2]
            break
        endif
        for i = 0 to 3
            new_r = s[0] + dir[0][i]
            new_c = s[1] + dir[1][i]
            if new_r < 0 or new_r >= maze_num or new_c < 0 or new_c >= maz
e_num or maze[new_r][
                new_c] == 1 or visited[new_r][new_c]
                continue
            endif
            q.append([new_r, new_c, s[2] + 1, qHead])
```

```
                qTail += 1
                visited[new_r][new_c] = 1
            qHead += 1
            next i
        endwhile
    endprocedure
```

Requirements met in puzzle 1

Requirements	Designed in which subroutine	How it is met
Problem illustration as straightforward as possible	initialise()	The maze will be visualised as a grid, which is relatively easy to understand
The user needs to think strategically in order to pass the game	initialise()	As the maze is generated randomly, the user will have to understand the principle in order to solve it
Principles behind games are logically strict	get_solution()	A Breadth-first search is used, which is guaranteed to find the best solution
Puzzles are designed to deepen the understanding of algorithms rather than broaden the knowledge	initialise()	Again, due to the randomness of the maze, a decent understanding is required to solve it

Puzzle 2: shortest path

The puzzle2 initialise has a similar function to that of the puzzle1: generating the graph and the player. For the same reason, a separate procedure is used in order to generate the graph multiple times. In order to make the graph look more random, the position of each node on the graph will also be generated on a random basis so that the user will not get tired of seeing the node in the same position each time he/she starts a new game. However, these positions cannot be entirely random as that will make the graph look messy and there is possibility that two nodes overlapping each other which can make the game hard to play. Therefore, the position of every node will be constrained in a certain region. This is achieved by dividing the whole screen into small squares.

```
procedure initialise()
    node_num = random(3, 30)
    # generate the position of every node
    num = sqrt(node_num) + 1
    sep = 500 / num
    i = 0
    j = 0
    minR = 500
    for k =0 to node_num
        start_x = int((j + 0.1) * sep)  # leave some blank space
        end_x = int((j + 0.9) * sep)
        start_y = int((i + 0.1) * sep)
        end_y = int((i + 0.9) * sep)
        circlePos = [random(int(start_x + sep * 0.2), int(end_x - sep * 0.2)),
                     random(int(start_y + sep * 0.2), int(end_y - sep * 0.2))]
        # make sure the node is not too small
        radius = min(circlePos[0] - start_x, end_x - circlePos[0], circlePos[1]
- start_y, end_y - circlePos[1])
        minR = min(radius, minR)
        node_list.append(new Node(circlePos, radius, k, DARKGREY))
        j += 1
        if j == num
            i += 1
            j %= num
        endif
    next k

    # generate the graph in a adjacency list
    graph = [node_num]
    for i =0 to node_num
        used = [node_num]
        used[i] = 1
```

```
            outDegree = random(1, int(node_num * 0.8))
            j = 0
            while j < outDegree
                n = random(0, node_num)
                if used[n]
                    continue
                endif
                used[n] = 1
                j += 1
                w = random(1, 100)
                node = node_list[n]
                node.set_weight(w)
                graph[i].append(node)
            endwhile
        next i

        # initialise the player
        n1 = node_list[random(0, node_num)]
        my_player =new Player2(n1.get_centre(), n1.num, [minR, minR], BLUE)
        visited_node.append(n1)
        all_sprites_group.add(my_player)
        while True
            n2 = node_list[random(0, node_num)]
            if n2.num != n1.num
                princess =new NPC(n2.get_centre(), n2.num, [n2.get_size(), n2.get_s
    ize()], RED)
                all_sprites_group.add(princess)
                princess_cor = n2.num
                break
            endif
        endwhile

    end procedure
```

The optimum solution in this level is calculated through the get_solution method and the Dijkstra's algorithm is used. As opposed to the first puzzle, which is solved using breadth-first search, but that algorithm cannot be applied to a weighted graph, hence Dijkstra's is used instead as it is easy to implement and does work on a weighted graph. The path may also need to be stored so that backtrack is possible.

```
    procedure get_solution()  # find the optimum solution of a problem
        solution_list = []
        visited = [0] * node_num
        dist = [INF] * node_num
        prev = [0] * node_num
        pq = PriorityQueue()
        nd = node_list[my_player.get_cor()]
        nd.set_weight(0)
        pq.put(nd)
        prev[nd.num] = -1
        dist[nd.num] = 0
        while not pq.empty()
            nd = pq.pop()
            if visited[nd.num]
                continue
            endif
            if nd.num == princess_cor
                solution = dist[nd.num]
                father = nd.num
                while True
                    solution_list.append(father)
                    if prev[father] == -1
                        break
                    endif
                    father = prev[father]
                endwhile
                solution_list.reverse()
                break
            endif
            visited[nd.num] = 1
            for i = 0 to len(graph[nd.num])
                n = graph[nd.num][i]
                if visited[n.num]
                    continue
                endif
                if dist[n.num] > dist[nd.num] + n.weight
```

```
                    dist[n.num] = dist[nd.num] + n.weight
                    prev[n.num] = nd.num
                    pq.enQueue(n)
                endif
            next i
        endwhile
    end procedure
```

A back button specific to this puzzle will also be included here to ebanle the user to move back when they make a mistake.

```
    procedure back()  // function of the back button
        if back_button.is_pressed()
            my_player.move_back()
        endif

    endprocedure
```

Requirements met in puzzle 2

Requirements	Designed in which subroutine	How it is met
Problem illustration as straightforward as possible	initialise()	A typical graph representation, using circles for nodes and line for edges
The user needs to think strategically in order to pass the game	initialise()	As the graph is generated randomly, the user will have to understand the principle in order to solve it
Principles behind games are logically strict	get_solution()	A Dijkstra's algorithm is used, which is guaranteed to find the shortest path here
Essential functions are implemented and available to the user	back()	It makes it easy for the user to undo a move

Puzzle 3: knapsack problem

Similarly, the initialise method is used to generate the bag and items in this puzzle. This time the initialise method is relatively simple compared to the previous two as the main focus of this puzzle is the mathematical principle used to solve the problem. The weight and volume of every element will still be generated randomly so that a user will be faced with different situations each time.

```
procedure initialise()
    item_num = random(4, 30)
    my_player = new Bag([250, 65], [100, 70], RED, random.randint(10, 1
00))
    all_sprites_group.add(my_player)
    num_x = int(sqrt(10 / 7 * item_num)) + 1
    num_y = int(sqrt(7 / 10 * item_num)) + 1
    sep_x = 500 / num_x
    sep_y = 350 / num_y
    i = 0
    j = 0
    for k = 0 to item_num-1
        start_x = int((j + 0.3) * sep_x)
        end_x = int((j + 0.7) * sep_x)
        start_y = 150 + int((i + 0.3) * sep_y)
        end_y = 150 + int((i + 0.7) * sep_y)
        item_pos = [start_x, start_y]
        size = [end_x - start_x, end_y - start_y]
        v = random(5, 50)
        w = random(1, 100)
        item = new Item(item_pos, size, BLACK, v, w)
        item_list.append(item)
        all_sprites_group.add(item)
        j += 1
        if j == num_x
            i += 1
            j = j mod num_x
        endif
    next k
endprocedure
```

Dynamic programming is the technique used here to calculate the solution. Obviously, this problem can be solved using a brute force method – by listing all possibilities of combining items in a bag and examine the value obtained each time, which will take exponential time. Therefore, dynamic programming is chosen as an alternative method, which is a pseudo-polynomial time algorithm for this problem. This means that in most of the case it will solve the problem in a reasonable amount of time. Similar to the last puzzle, the actual combination of items may also need to be stored in order to be displayed on the screen.

```
procedure get_solution()  # find the optimum solution of a problem
    w = [0] * (item_num + 1)
    v = [0] * (item_num + 1)
    for i = 1 to item_num
        w[i] = item_list[i - 1].get_weight()  # i-
1 is because index starts from 0 in the item_list
        v[i] = item_list[i - 1].get_volume()
    next i
    dp = [my_player.volume + 1,item_num + 1]
    choices = [my_player.volume + 1, item_num + 1]
    # store how items are selected for dp[i][j]
    for i =1 to item_num
        for j =1 to my_player.get_volume()
            dp[i][j] = dp[i - 1][j]
            choices[i][j] = choices[i - 1][j]
            if j >= v[i] and dp[i - 1][j - v[i]] + w[i] > dp[i - 1][j]
                dp[i][j] = dp[i - 1][j - v[i]] + w[i]
                choices[i][j] = choices[i - 1][j - v[i]] + [i]
            endif
        next j
    next i
    solution = dp[item_num][my_player.get_volume()]
    solution_list = choices[item_num][my_player.get_volume()]

endprocedure
```

Items should be highlighted to make it easier for the user to see what they have selected, and here is the pseudocode:

```
procedure item_highlight()  // show items that are selected
    for item in Puzzle3.item_list
        added = item.clicked()
        if item.selected
            item.highlight(GREEN)
        endif

        // update the value of the bag
        if leftclicked
            bag.weight += item.get_weight()
            bag.volume -= item.get_volume()

        else if rightclicked
            bag.weight -= item.get_weight()
            bag.volume += item.get_volume()
        endif

    next item

endprocedure
```

Requirements met in puzzle 3

Requirements	Designed in which subroutine	How it is met
Problem illustration as straightforward as possible	highlight()	Items that have been selected will be highlighted which can be useful when the user is making the decision.
The user needs to think strategically in order to pass the game	initialise()	As the combination of items is generated randomly, the user will have to understand the principle in order to solve it
Principles behind games are logically strict	get_solution()	A dynamic programming is used, which is typically how the knapsack problem is solved

Testing design

This game will be tested at various stages, each with a different aim. At the first stage, the game will be checked against the most important requirements (indicated by "must" in the success criteria) to make sure that it has all the basic elements running correctly. After the first test, the game is able to run properly with all the key features listed in the requirement.

No.	Requirement(s)	Design	Expected output
	Level1		
1.	Problem illustration as straightforward as possible	Maze generation	Maze is correctly generated and easy to operate on
2.	Player movement	Arrow keys to move the player	Player moves accordingly
3.	Clear instruction on how to play the game	Wining text displayed	Text are displayed when the game starts
4.	All information required is displayed	Steps taken change as player moves	As required by the design
5.	Essential functions are implemented and available to the user	Retry button working	The game starts again with the same setting, e.g. the same map
6.	Essential functions are implemented and available to the user AND The user needs to think strategically in order to pass the game	Restart button working	The game starts again with a different setting
7.	Essential functions are implemented and available to the user	Display solution button working	Solution displayed
8.	Game running correctly and logically	The player is moved to the boundary of the maze	The player will not move any more
	Algorithm test		
9.	N/A	initialise method generates the maze and the player	The maze, player and target are ready when the game starts
10.	Principles behind games are logically strict	get_solution method calculates the minimum steps required for the player to reach the target	Manual trace can be used to check the correctness of the algorithm
Level2			
11.	Problem illustration as	A graph made of edges	As required in the

	straightforward as possible	and weights displayed	design
12.	Player movement	Player able to move	Player moves when a node is clicked
13.	All information required is displayed	Path taken displayed	Trajectory is shown when the player moves
14.	Game running correctly and logically	Player cannot move to a node that is connected	Player will not move when the user clicks a node that is not connected
15.	Clear instruction on how to play the game	Game information displayed	Text are displayed when the game starts
16.	Essential functions are implemented and available to the user	Edges displayed when mouse is over	As required in the design
17.	Essential functions are implemented and available to the user	Retry button working	The game starts again with the same setting, e.g. the same graph
18.	Essential functions are implemented and available to the user AND The user needs to think strategically in order to pass the game	Restart button working	The game starts again with a different setting,
19.	Essential functions are implemented and available to the user	Display solution button working	Solution displayed
20.	Essential functions are implemented and available to the user	Back button working	To undo the movement, click the undo button
21.	Essential functions are implemented and available to the user	All edges button working	All edges are displayed when the button is clicked
	Algorithm test		
22.	N/A	initialise method generates the maze and the player	The maze, player and target are ready when the game starts
23.	Principles behind games are logically strict	get_solution method calculates the shortest path required for the player to reach the target	Manual trace can be used to check the correctness of the algorithm
Level3			
24.	Problem illustration as	Bags and items are	Text are displayed when

	straightforward as possible	displayed and easily identified	the game starts
25.	All information required is displayed	Weight and volume underneath	Text are displayed when the game starts
26.	Clear instruction on how to play the game	Game instruction shown	Text are displayed when the game starts
27.	Clear instruction on how to play the game	Game information changes dynamically	correct information is displayed when the game progresses
28.	Game running correctly and logically	Clicking the same item twice	No change should happen as the same item cannot be chosen twice
29.	Essential functions are implemented and available to the user	Retry button working	The game starts again with the same setting, e.g. the bag is used
30.	Essential functions are implemented and available to the user AND The user needs to think strategically in order to pass the game	Restart button working	The game starts again with a different setting,
31.	Essential functions are implemented and available to the user	Display solution button working	Solution displayed
32.	Essential functions are implemented and available to the user	To select items	Left click to select items and selected items are highlighted in green
33.	Essential functions are implemented and available to the user	To deselect items	Right click to deselect items and green lines will disappear
	Algorithm test		
34.	N/A	initialise method generates the bag and items	The bag and items are ready when the game starts
35.	Principles behind games are logically strict	get_solution method calculates the largest weight possible for a given volume	Manual trace can be used to check the correctness of the algorithm

Secondly, the game will be tested again according to the “should” requirements, as these ensure a good user experience while playing and that the objective of this game — illustrating abstract and complex computational algorithms as vividly as possible is achieved.

Finally, another check will be carried out against requirements followed by “could”. These are additional features that the game could have so that it is more interesting in terms of playability. All the basic functionalities and objectives will have been achieved by the previous tests.

Developing and testing

Following on from the Design section, where the problem is decomposed into many small parts that can be solved individually, the actual implementation of this problem will be mostly based on how it is divided into small sections. There will be four main stages in this development, and every stage is broken down further into small tasks.

At the first stage, it will include some implementations of the general UI framework as described in the general design of puzzles, this includes game instructions, game progress and buttons, etc. It will also incorporate all elements in the first puzzle as it is designed.

Following on from the first stage, the second and the third stage will be centred around the second and the third puzzle, based on how they are designed. Finally, all puzzles are integrated together by implementing the puzzle selection interface, which is the main objective of the last stage.

Prototypes will be developed and tested against the requirements, and any improvements can be if necessary.

Stage1: puzzle1

According to the top down design section of the first puzzle, this puzzle is divided into eight parts, three of which are dependent on previous steps. The order in which these problems are addressed are not particularly important if there is no dependency between them.

Maze generation

This game is based on a maze to run and it is required that the game needs to have straightforward illustrations, therefore, the first step to be implemented is to generate the maze, player and the target.

The maze will be implemented as a 2d array where 1s will represent free squares and 0s will be used when a square is blocked, i.e. the player cannot move onto it, and it is shown in black on the image below. Positions of blocked squares will be randomly generated so that the game can be played many times. The position of the player and the target will also be randomly generated, and an object of the player and the target will be generated on the designated position. Moreover, the size of the maze will be generated randomly. This is due to requirement 13 that a user needs to think strategically to solve the puzzle, which is why randomness is necessary.

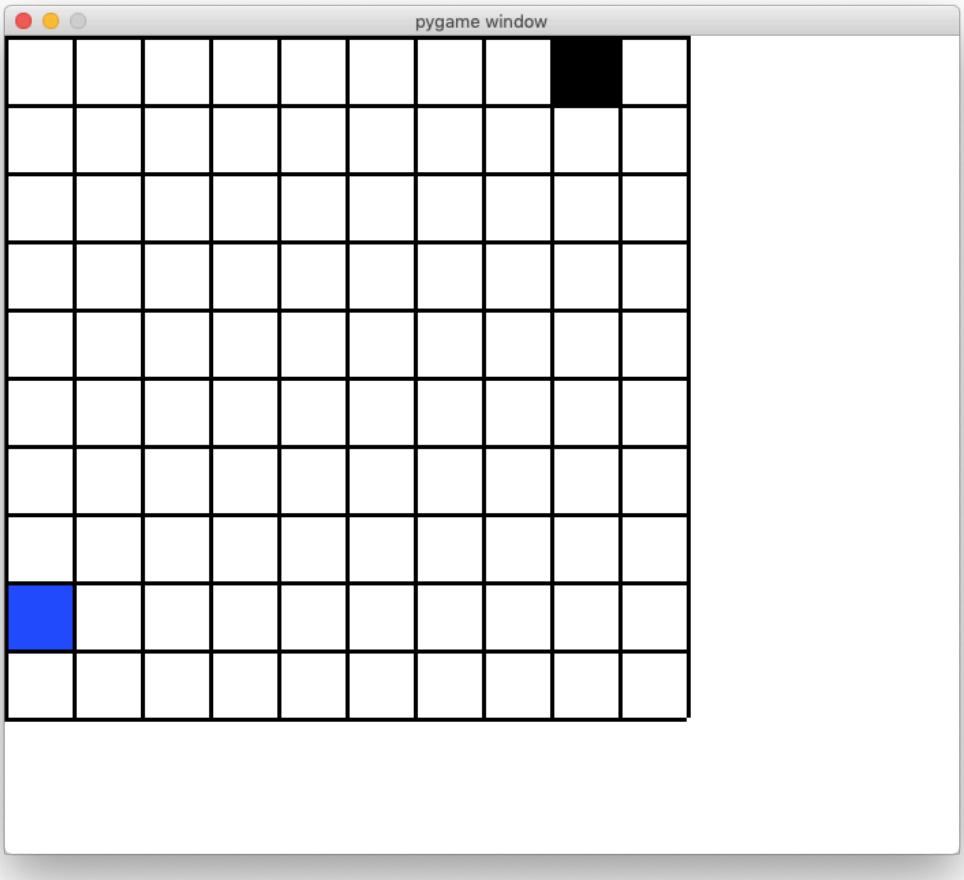
This is achieved by the following code:

```
def generate_maze():  # in addition to generate the maze and add it to groups, also
returns the myPlayer object
    # 1=wall 2=player 3=princess
    global maze
    nMazeNum = 30
    maze = [[0] * nMazeNum for i in range(nMazeNum)]
    princessPos = [0, 0]
    nSpecialElement = random.randrange(0, int(nMazeNum * nMazeNum * 0.6))  # randrange
[a,b)
    playerCor = random.randrange(0, nMazeNum * nMazeNum)
    maze[playerCor // nMazeNum][playerCor % nMazeNum] = 2
    while True:
        princessCor = random.randrange(0, nMazeNum * nMazeNum)
        if princessCor != playerCor:
            break
    maze[princessCor // nMazeNum][princessCor % nMazeNum] = 3
    nSpecialElement -= 2
    print("nSpecialElemen=", nSpecialElement, "nMazeNum=", nMazeNum)
    for i in range(nSpecialElement):
        while True:
            wallCor = random.randrange(0, nMazeNum * nMazeNum)
            if wallCor != princessCor and wallCor != playerCor:
                break

        maze[wallCor // nMazeNum][wallCor % nMazeNum] = 1
    for i in range(nMazeNum + 1):  # +1 in oder to add grid lines of both ends
        sideLength = 500 / nMazeNum
        wall = Tile((0, i * sideLength), (500, 3), BLACK)  # adding grid lines
        wall_group.add(wall)
        all_sprites_group.add(wall)
        for j in range(nMazeNum + 1):
            if (i == 0):
                wall = Tile((j * sideLength, 0), (3, 500), BLACK)
                wall_group.add(wall)
                all_sprites_group.add(wall)

            if i == nMazeNum or j == nMazeNum:  # if it is the fringe of the maze, then
go on
                continue

            if (maze[i][j] == 1):
```

```
                tile = Tile([j * sideLength, i * sideLength], [sideLength + 0.8,
sideLength + 0.8], BLACK)
                # adding 0.8 is not a very good way but makes the maze look better
                tile_group.add(tile)
                all_sprites_group.add(tile)

            elif maze[i][j] == 2:
                size = [sideLength * 0.4, sideLength * 0.4]  # the size of the
player will make up two fifths of a tile
                myPlayer = Player((j * sideLength + sideLength * 0.3, i * sideLength
+ sideLength * 0.3), [i, j], size,
                                  BLUE)  # player is centred

            elif maze[i][j] == 3:
                size = [sideLength * 0.4, sideLength * 0.4]
                princess = Tile((j * sideLength + sideLength * 0.3, i * sideLength +
sideLength * 0.3), size, RED)
                all_sprites_group.add(princess)
                princessPos = [i, j]
```

Testing

The program is then run to be tested. It does give the expected output and the user should be able to understand the maze at first glance as specified in the first test design, when the size of the maze is not particularly large. As this is the first version of the maze, the maze structure is relatively simple.



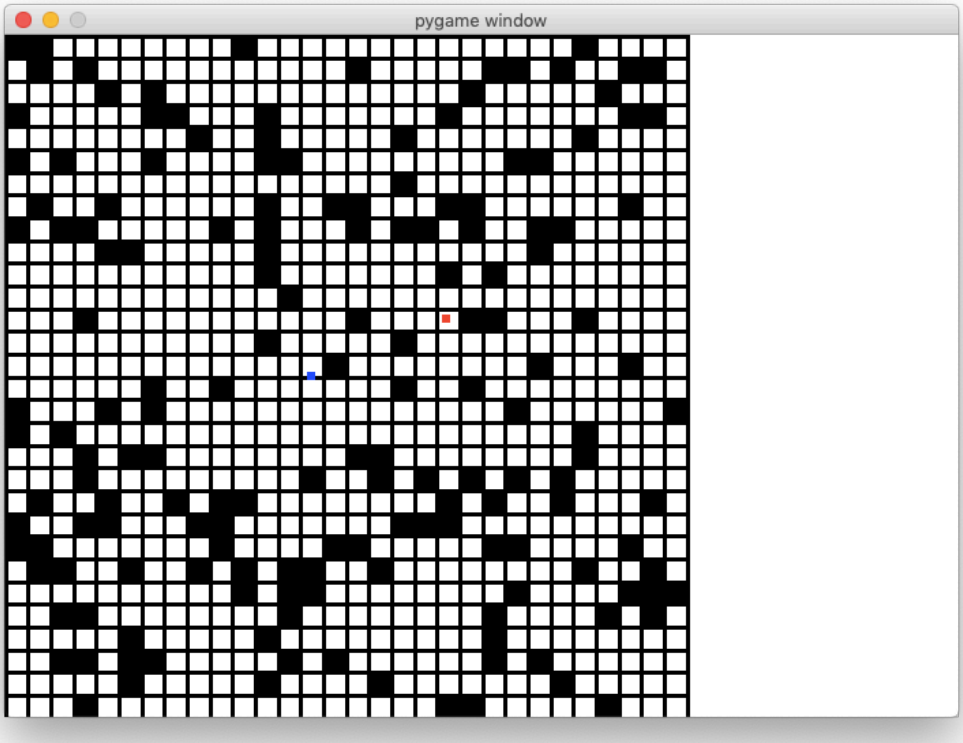
Player movement

Secondly, the ability of the player of to make movements, which is the second requirement, is implemented as the method of the Player1 class, considering that this is one of the behaviours that the player1 should have. This is achieved by calling a method of the player called tracking_event in the main program loop that will track if there is any input from the user and will then call the move method with the correct parameters. For example, if the down key is pressed, then the coordinate of the square below the player will be passed to the move method. In addition, according to the requirement that the game should run logically and correctly, the player should not be able to make invalid moves, therefore it will be checked in the move method that the player is not moving to blocks that are not reachable.

```
def tracking_event(self,keys):
    if keys == pg.K_RIGHT:
        self.move([self.cor[0], self.cor[1]+1])
    elif keys == pg.K_LEFT:
        self.move([self.cor[0], self.cor[1] -1])
    elif keys == pg.K_UP:
        self.move([self.cor[0]-1, self.cor[1]])
    elif keys == pg.K_DOWN:
        self.move([self.cor[0]+1, self.cor[1]])


def move(self,end_cor):
    new_r=end_cor[0]
    new_c=end_cor[1]
    flag = (0 <= new_r < Puzzle1.maze_num) and (0 <= new_c < Puzzle1.maze_num) and
(Puzzle1.maze[new_r][new_c] != 1)
    # check if the player reaches the boundary and if the block is reachable
    if flag:
        # find the new centre of the player
        newCentre = Puzzle1.tile_list[new_r][new_c].get_centre()
        newX = newCentre[0] - self.size[0] / 2
        newY = newCentre[1] - self.size[1] / 2
        # change the attributes of the player
        self.rect.x = newX
        self.rect.y = newY
        self.cor[0] = new_r
        self.cor[1] = new_c
        self.step += 1
```

Testing

Again, the program is tested against the second test design. However, this time, when the player is moving a maze with a large size, a problem arises: The player no longer stays at the centre of each tile as it moves further in one direction.



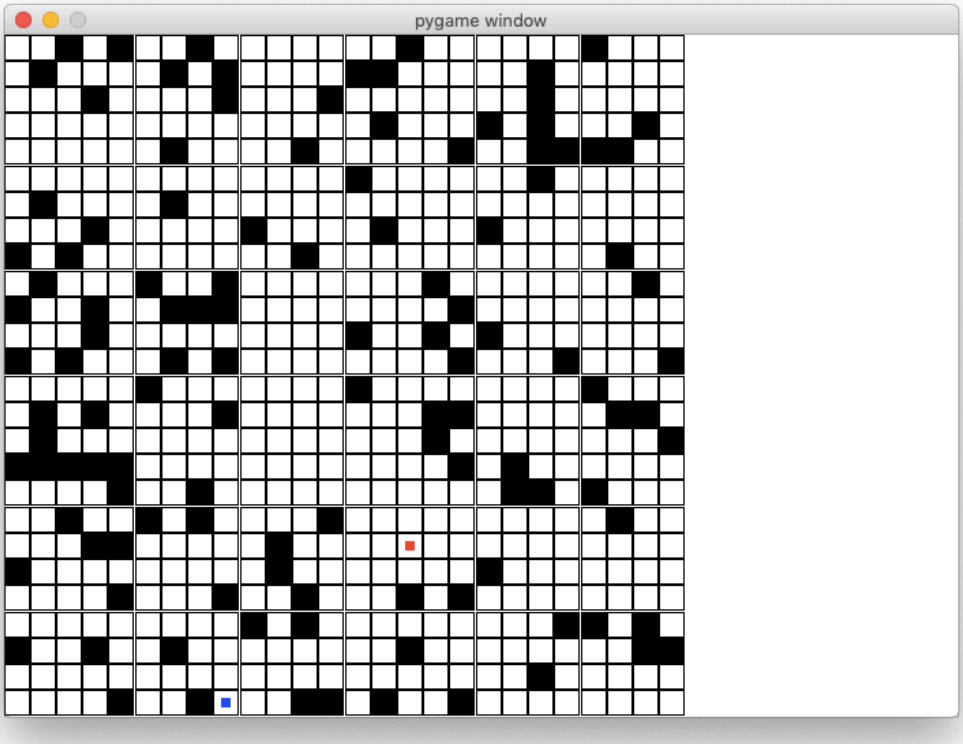
As one can see in the screen shot that the player (the blue square) moves off the centre of a tile, which may cause confusion to the user as to which square is the player currently in and thus worsen the experience of a user.

This problem is due to the definition of walls. Previously, walls are defined as long black lines, and the way the player moves is that it will move a fixed distance each time calculated from the separation between black lines, which might be rounded. When the player moves multiple times in one direction, those small errors start to accumulate and cause the player to move off the centre.

In order to resolve this issue, I need to go back to the place where I defined walls as long black lines. Instead of doing that, I defined each tile in the maze as an object itself. In this way, the coordinates to which the player is moving can be attributes embedded in each tile, and this ensures that the player will always stay in the centre no matter which tile it moves to.

```
Puzzle1.tile_list = [[Tile([0,0],[0,0])] * Puzzle1.maze_num for i in
range(Puzzle1.maze_num)]


for i in range(Puzzle1.maze_num):
    for j in range(Puzzle1.maze_num):
        t = Tile((j * sideLength, i * sideLength), (sideLength, sideLength))
        if Puzzle1.maze[i][j] == 1:
            t.set_color(0)
        Puzzle1.tile_list[i][j] = t
```

And this does prove to resolve the issue:



Information display

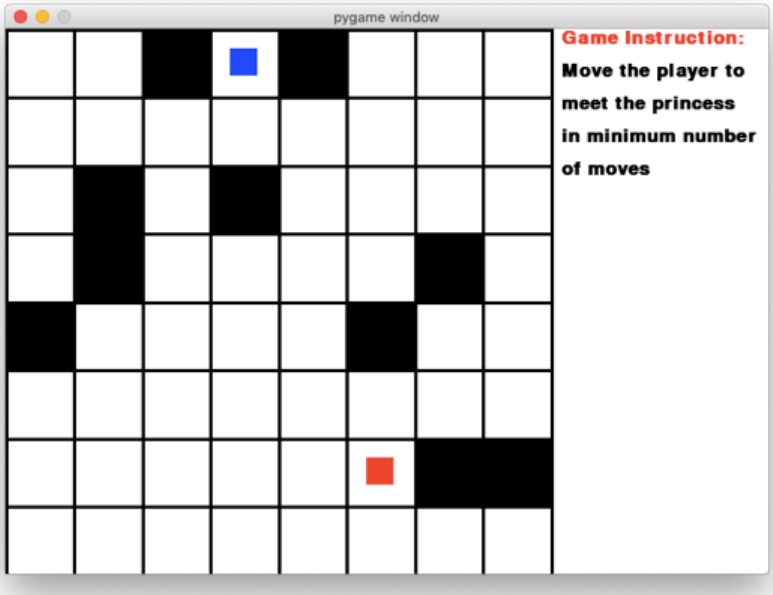
Thirdly, the game information including instructions, progress, etc will be displayed, as required by requirement 3 and 6, in the area on the right-hand side of the game interface.

This will be implemented as a procedure and will be called in the main loop to keep it displayed on the screen.

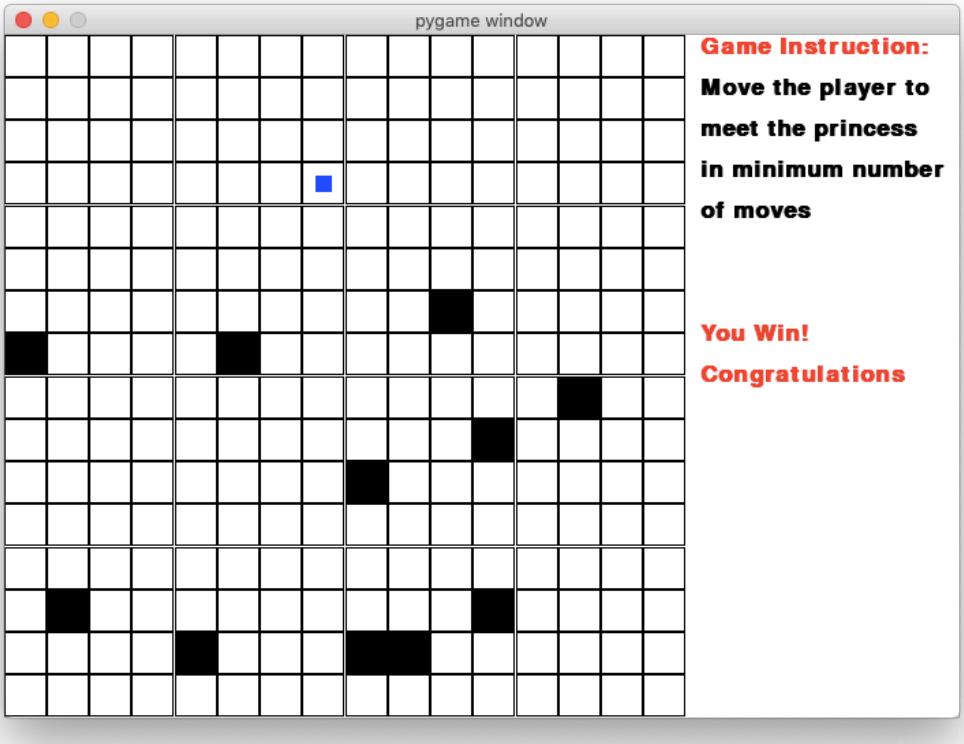
```
def display_info(self):
    # game instruction
    gameInstruction = []
    gameInstruction.append(font.render("Game Instruction:", True, RED))
    gameInstruction.append(font.render("Move the player to", True, BLACK))
    gameInstruction.append(font.render("meet the princess", True, BLACK))
    gameInstruction.append(font.render("in minimum number", True, BLACK))
    gameInstruction.append(font.render("of moves", True, BLACK))
    for i in range(len(gameInstruction)):
        screen.blit(gameInstruction[i],[500+10,i*30])


    # game information
    if self.my_player.get_cor()==Puzzle1.princess_cor and
self.my_player.get_step()==self.solution:
        screen.blit(font.render("You Win!", True, RED), [500 + 10, 200 + 10])
        screen.blit(font.render("Congratulations",True,RED),[500+10,200+10+30])
    elif self.my_player.get_cor()==Puzzle1.princess_cor:
        screen.blit(font.render("Well done!", True, RED), [500 + 10, 200 + 10])
        screen.blit(font.render("Try to do it with", True, RED), [500 + 10, 200 + 10
+ 30])
        screen.blit(font.render("fewer moves", True, RED), [500 + 10, 200 + 10 +
30*2])
```

Testing

The screen shot below illustrates the effect of adding instructions to the game.



Moreover, when the player finishes the game successfully, a piece of text indicating that the player has won the game will be displayed, which matches perfectly with the test criterion3.



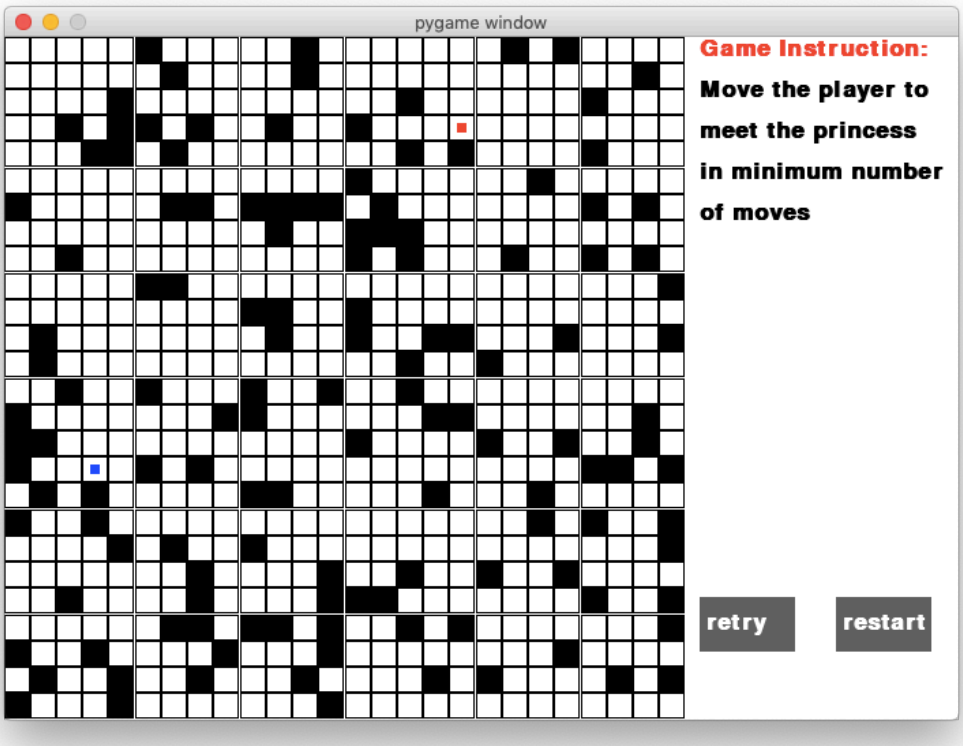
Button functions

According to requirement 12, essential functions should be made available to the user. Specifically, this is implemented as buttons in the game which provide different functionalities such as restart the game. Buttons are defined as a separate class, the methods of which provide the functionalities.

```
class Button(object):
    def __init__(self, pos, size, color, word):
        self.image = pg.Surface(size)
        self.rect = self.image.get_rect(topleft=pos)
        self.image.fill(color)
        self.word = word
        self.size = size

    def is_over(self):  # returns whether the mouse is over the button, not necessarily
a click
        mouse_pos = pg.mouse.get_pos()
        return self.rect.x < mouse_pos[0] < self.rect.x+self.size[0] and self.rect.y <
mouse_pos[1] < \
            self.rect.y+self.size[1]

    def is_pressed(self):  # returns if the button is pressed
        return self.is_over() and pg.mouse.get_pressed()[0]  # mouse needs to be over a
certain button

    def display(self):  # display word and the button on the screen
        screen.blit(self.image, self.rect)
        screen.blit(font.render(self.word, True, WHITE), [self.rect.x+5,
self.rect.y+10])

    def switch(self, color):  # switch the color of the button
        self.image.fill(color)

    def update(self):
        if self.is_over():
            self.switch(DARKGREY)
        else:
            self.switch(GREY)
        self.display()
```

Testing

Functions of buttons are also tested. Here is a glance at two buttons displayed on the screen. The retry button enables one to reset the position of the player without changing the maze while the restart button changes the setting of the maze so that users can try the puzzle with different sizes.

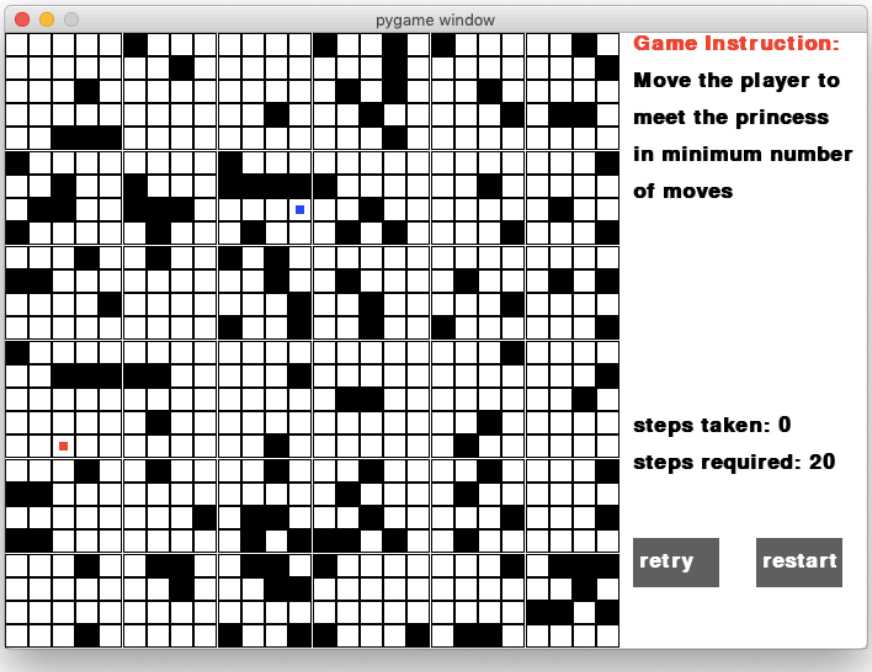


According to testing design No. 5 and 6, two of these buttons need to be working as expected. It is not easy to show how these two buttons work with screenshots, but they have been tested and they do indeed give the expected output.

Solution in BFS

Finally, in accordance with requirement 3, the correct solution to the puzzle will be calculated once the maze has been generated. BFS is used here as justified in the design. The algorithm used here to find the minimum number of moves is breadth-first search, which is commonly used in finding an optimum solution. A queue is used to implement the search. The numerical answer will also be displayed in the game information section so that users know what is the optimum solution they are looking for.

```
def Bfs(startPpos, endPos):
    global maze, nMazeNum
    q = deque()  # every element in q is a list of three integers s[0]: row num,
s[1]:column number, s[2]: number of steps
    visited = [[0] * nMazeNum for i in range(nMazeNum)]
    dir = [[0, 1, 0, -1], [1, 0, -1, 0]]
    q.append([startPpos[0], startPpos[1], 0])
    while q:
        s = q.popleft()
        print("element in queue", s)
        if [s[0], s[1]] == endPos:
            return s[2]
        for i in range(4):
            newR = s[0] + dir[0][i]
            newC = s[1] + dir[1][i]
            if newR < 0 or newR >= nMazeNum or newC < 0 or newC >= nMazeNum or
maze[newR][newC] == 1 or visited[newR][
                newC]:
                continue
            q.append([newR, newC, s[2] + 1])
            visited[newR][newC] = 1
    return -1
```

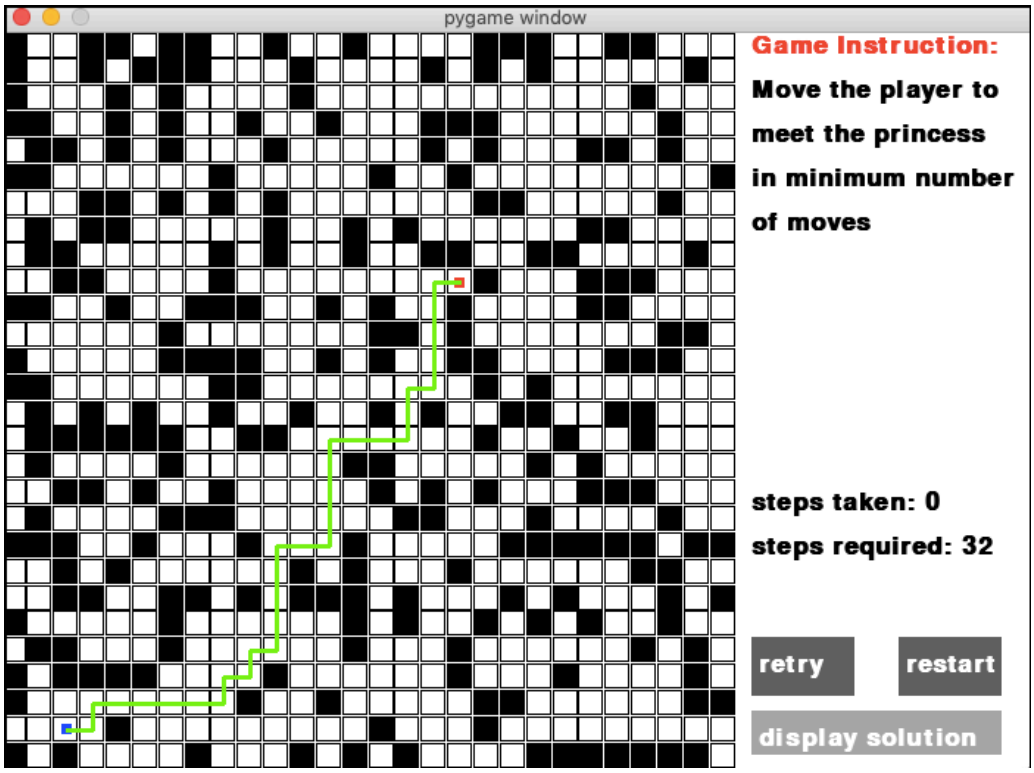


Testing

However, by testing this, one can see that only the number of the optimum solution will be available to the user, but not how this number is found, which makes it harder for a user to trace through the actual shortest path. In order to reflect the requirement No.16 that the solution straightforward for the user to follow, a graphical solution will be displayed to the user via the solution button. And this is done by recording the actual path, in other words, update the previous node (named father node in the code) of each node while the algorithm is calculating the solution.

```
def get_solution(self):
    self.solution = -1
    self.solution_list = []
    q = []
    qHead=0
    qTail=0
    # every element in q is a list of three integers s[0]: row num, s[1]:column number;
    # s[2]: number of steps, s[3]: father
    visited = [[0] * Puzzle1.maze_num for i in range(Puzzle1.maze_num)]
    dir = [[0, 1, 0, -1], [1, 0, -1, 0]]
    startPos = self.my_player.get_cor()
    endPos = Puzzle1.princess_cor
    q.append([startPos[0], startPos[1], 0, -1])
    qTail += 1
    # bfs implementation
    while qHead!=qTail:
        s = q[qHead]
        if [s[0], s[1]] == endPos:
            self.solution=s[2]
            father = qHead
            while True:
                self.solution_list.append([q[father][0],q[father][1]])
                father=q[father][3]
                if father==-1:
                    break
            self.solution_list.reverse()
            break
        for i in range(4):
            new_r = s[0] + dir[0][i]
            new_c = s[1] + dir[1][i]
            if new_r < 0 or new_r >= Puzzle1.maze_num or new_c < 0 or new_c >=
Puzzle1.maze_num or Puzzle1.maze[new_r][
                new_c] == 1 or visited[new_r][new_c]:
                continue
            q.append([new_r, new_c, s[2] + 1,qHead])
            qTail+=1
            visited[new_r][new_c] = 1
        qHead += 1
```

And this is the how the path is displayed when the display solution button is held. The green path is one if the correct paths that will give the steps required.



Review

Until now the puzzle1 has been completed with 5 steps. And this stage is broken down into these steps to reflect the following requirements:

- Player movement
- All information required is displayed
- Game running correctly and logically
- Problem illustration as straightforward as possible
- Essential functions are implemented and available to the user
- Solutions are straightforward for players to follow

There are also some changes and improvements that will be included in the next two puzzles, in order to improve the game logic, user experience, etc.

1. Each puzzle will be defined as a class and all of its behaviours, such as finding the solution, displaying game instructions, will be implemented as its methods. There are also some attributes that will be included such as a list of tiles in the first puzzle. Furthermore, in order to make the relation between different puzzles clear, each puzzle class will inherit a same superclass that acts like a virtual class.
2. Whilst developing the first puzzle, a problem that the player can move off the centre inspired me to define tile as a class. In the next puzzle that I am going to develop, which is to do with graph theory, where nodes will be used, the node will be defined as a class as well. This proves to quite an efficient way based on using tile as a class in the first puzzle.
3. Displaying the solution graphically in addition to the numerical answer to the puzzle could give the user more insight into how the problem can be solved and will be used in the next two puzzles as well.
4. Buttons will turn light grey when the mouse is over, which is naturally how a button behaves. This can give the user better experience while solving the puzzle, and will be used in future development as well.

Stage2: puzzle2

Again, the second puzzle is also divided into several steps in the design section. As each puzzle has a lot of features in common, steps that are designed to complete the stage will also have some similarities. Again, there is not a particular order in which different steps need to be completed, but I would follow the order in which the problem is broken down.

Graph generation

Graph, player and the target will need to be generated first so that further operations on them are possible. By the review from stage1, I have decided to implement puzzle2 as a class and it will inherit the super class Puzzle, just like the Puzzle1 class does. It is natural, after defining it as a class, to include the graph generation procedure as a method of class Puzzle2. The initialise method also overrides the method in the class Puzzle, so that polymorphism can be used to make the code more flexible. It is also part of the requirement that the idea of OO needs to be used in the program so that it can be easily managed and updated in the future.

An adjacency list will be used to store the graph as the number of edges is randomly generated and therefore the graph could be sparse, in which case an adjacency can be useful. The position of the player and the target will both be random, similar to how it is done in the first puzzle.

The following code is used to generate the graph:

```
def initialise(self):
    Puzzle2.node_num=random.randrange(3, 30)
    # generate the position of every node
    num = int(math.sqrt(Puzzle2.node_num)) + 1
    sep=500/num
    i = 0
    j = 0
    minR=500
    for k in range(Puzzle2.node_num):
        start_x=int((j+0.1)*sep)  # leave some blank space
        end_x=int((j+0.9)*sep)
        start_y=int((i+0.1)*sep)
        end_y=int((i+0.9)*sep)
        circlePos=[random.randint(int(start_x+sep*0.2),int(end_x-
sep*0.2)),random.randint(int(start_y+sep*0.2),int(end_y-sep*0.2))]
        # make sure the node is not too small
        radius=min(circlePos[0]-start_x,end_x-circlePos[0],circlePos[1]-start_y,end_y-
circlePos[1])
        minR=min(radius,minR)
```

```
        Puzzle2.node_list.append(Node(circlePos,radius,k,DARKGREY))
        j+=1
        if j == num:
            i += 1
            j %= num

    # generate the graph in a adjacency list
    Puzzle2.graph = [[] for i in range(Puzzle2.node_num)]
    for i in range(Puzzle2.node_num):
        used = [0] * (Puzzle2.node_num + 1)
        used[i]=1
        outDegree = random.randrange(1, int(Puzzle2.node_num * 0.8))
        j=0
        while j < outDegree:
            n = random.randrange(0, Puzzle2.node_num)
            if used[n]:
                continue
            used[n]=1
            j+=1
            w = random.randrange(1, 100)
            node=Puzzle2.node_list[n]
            node.set_weight(w)
            Puzzle2.graph[i].append(node)

    # initialise the player
    n1=Puzzle2.node_list[random.randrange(0, Puzzle2.node_num)]
    self.my_player=Player2(n1.get_centre(),[minR,minR],BLUE, n1.num)
    Puzzle2.visited_node.append(n1)
    all_sprites_group.add(self.my_player)
    while True:
        n2=Puzzle2.node_list[random.randrange(0, Puzzle2.node_num)]
        if n2.num!=n1.num:
            princess=NPC(n2.get_centre(),[n2.get_size(),n2.get_size()],RED, n2.num)
            all_sprites_group.add(princess)
            Puzzle2.princess_cor=n2.num
            break
```

Up till now the graph has been generated, but it will not be shown until the next step, and therefore the test will be carried out in the next step, edges and nodes display.

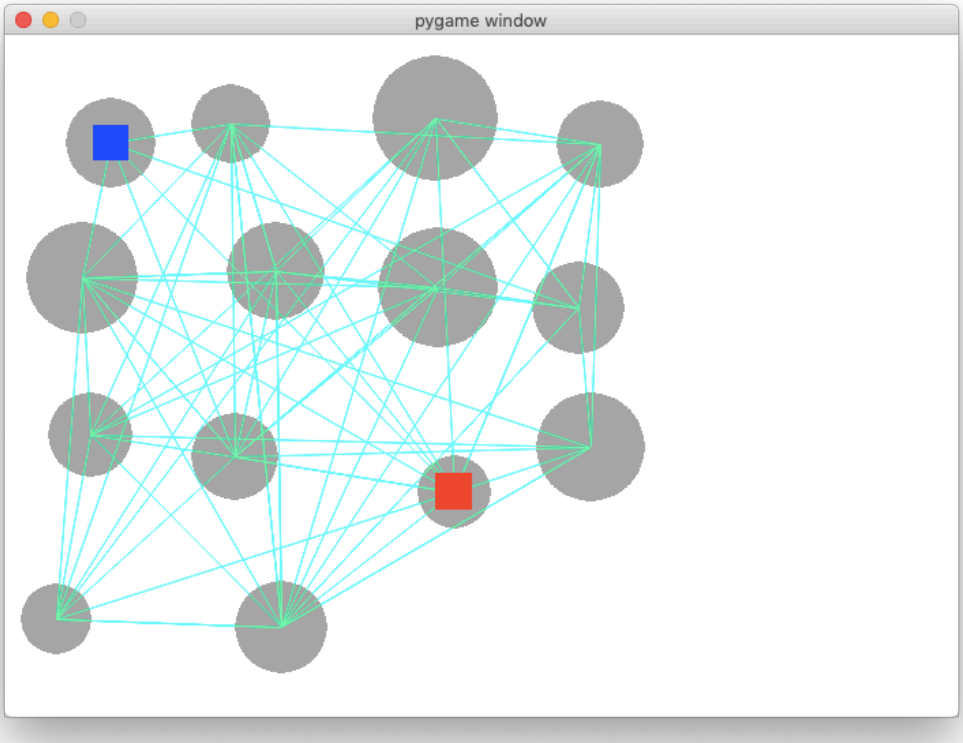
Edges and nodes display

The unique part of this puzzle2 is that it involves an essential graph element, i.e. edges and nodes. Therefore, it is important to draw these edges and nodes properly on the screen so that a user can not only get a better understanding of what a graph looks like, but also able to play the game easily, which is required by the 10th requirement. Nodes are drawn with the following method.

```
def draw_nodes():
    for i in range(Puzzle2.node_num):
        n=Puzzle2.node_list[i]
        pg.draw.circle(screen,n.get_color(),n.get_centre(),n.get_size())
```

Testing

Edges are, by definition, connections between different nodes. So, one would naturally draw a line between the centres of two nodes. Whilst it is easy to draw edges by hand when there is a small number of them, problems arise when the number of edges increase when one tries to sketch all of them, below is what it would look like.

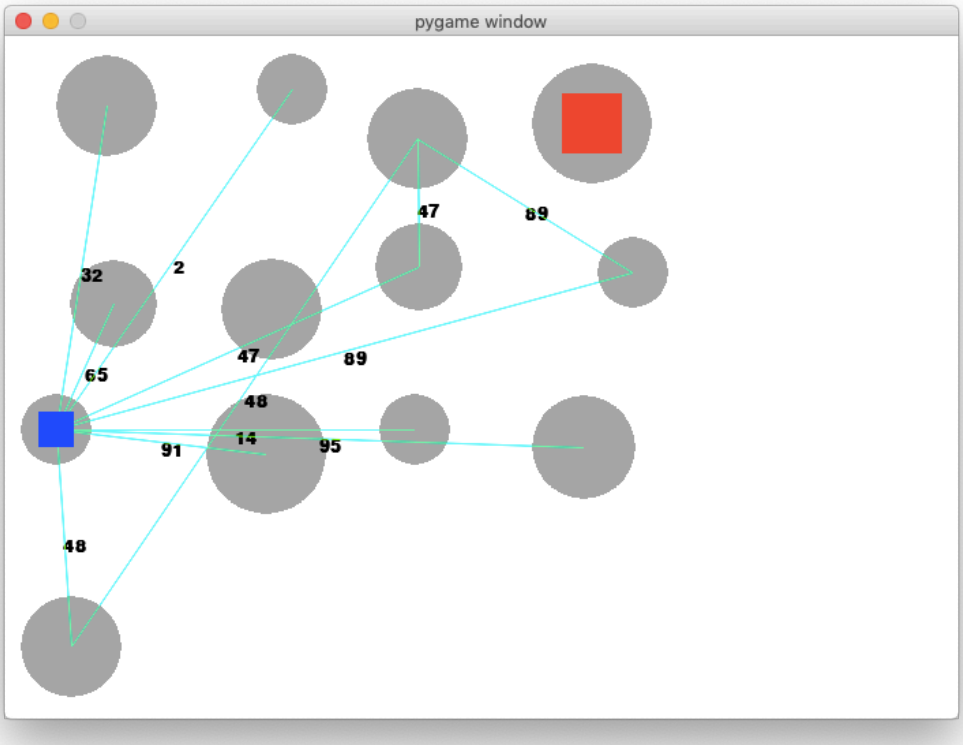


When tested, it is obvious that edges are too close together, and it is too hard for the player to figure which edges are connected to one node, and this also does not meet the requirement that this game should provide good user experience and problem illustration as people can get confused as to what this graph is about. Therefore, in order to solve this problem, I tried to display edges only when the player is in that node, or when the mouse is placed over a particular node. This can also improve the usability feature of the game.

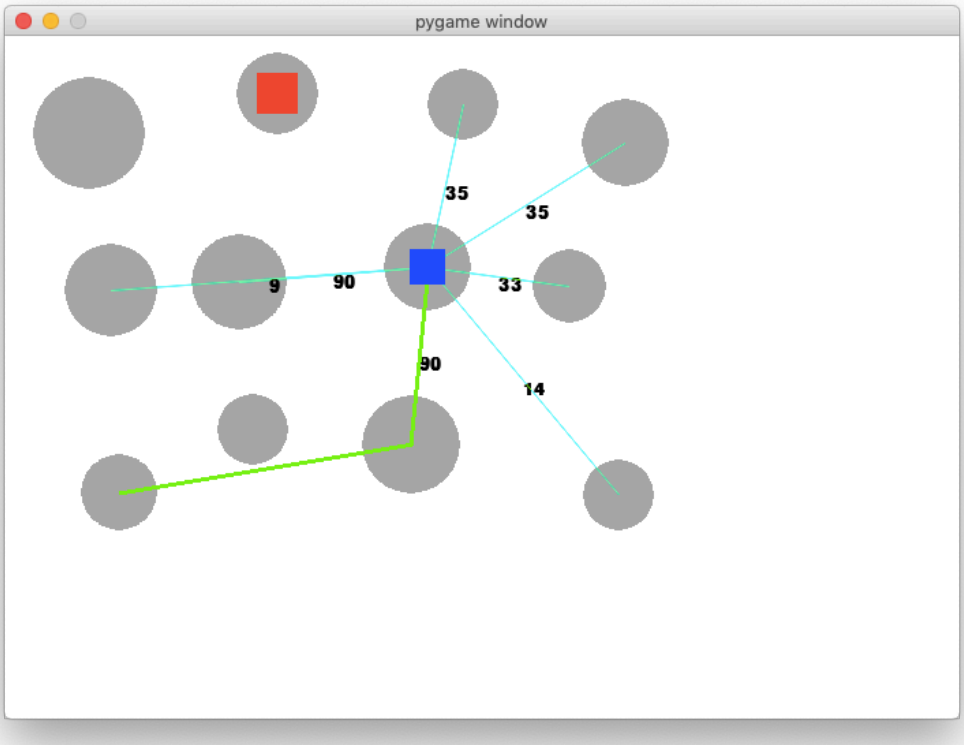
```
def draw_edges(self):  # draw the edge if the player is on the node or if the
mouse is over
    for n in Puzzle2.node_list:
        if n.is_mouse_over():
            self.draw_edges_from_node(n)

    n=Puzzle2.node_list[self.my_player.get_cor()]
    self.draw_edges_from_node(n)
```

Again, in order to illustrate the problem clearly, it is also important to consider carefully how the weights of each edge will be displayed, though this can be easily done by hand when the number of edges is small. Considering the overlapping between edges, I have decided to display the weight of each edge at the middle point of each edge so that when a user can always refer to the middle point when he/she is looking for the weight of the edge.

This is the final display of edges with weights on them.



This game is tested by playing it again and another problem is found. Users might find it hard to remember the path they have already taken, in other words, which nodes they have already visited, especially as there is a long way from the source to the destination. It might be useful to highlight the path that has been taken by the player so that it is easier for the user to make further decisions.



It can be seen that the path taken by the blue square is highlighted in light green and this makes it easy for the user to refer to what path they have taken. This is also a reflection of requirement 10 that the game can run smoothly and provides good user experience.

The game now gives expected output and passes the test 10, 12,14 and 21.

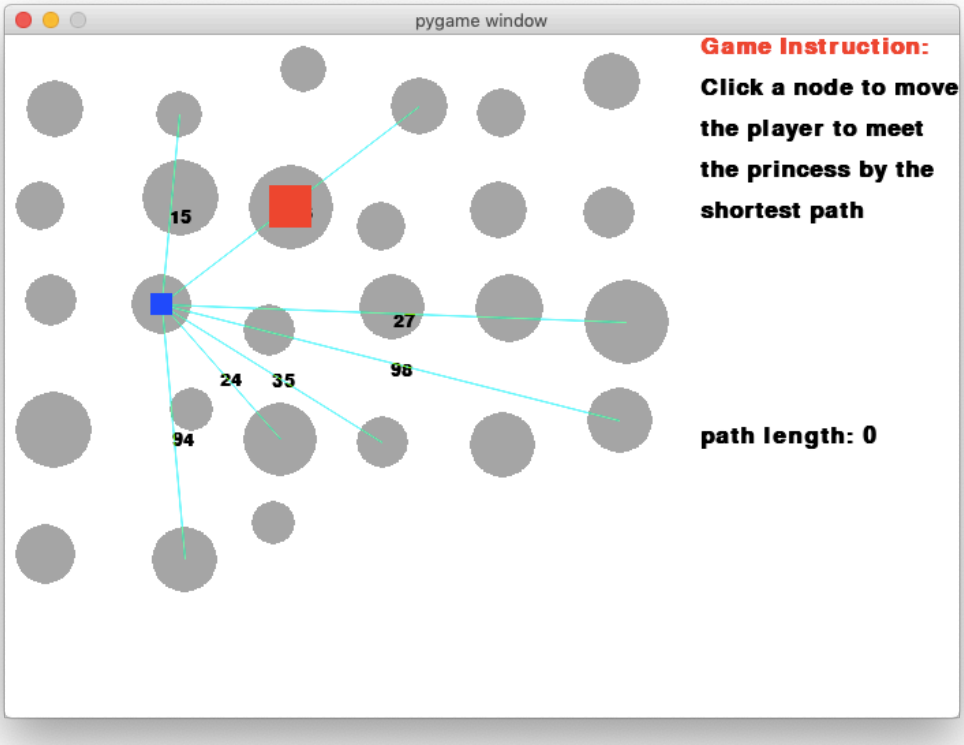
Information display

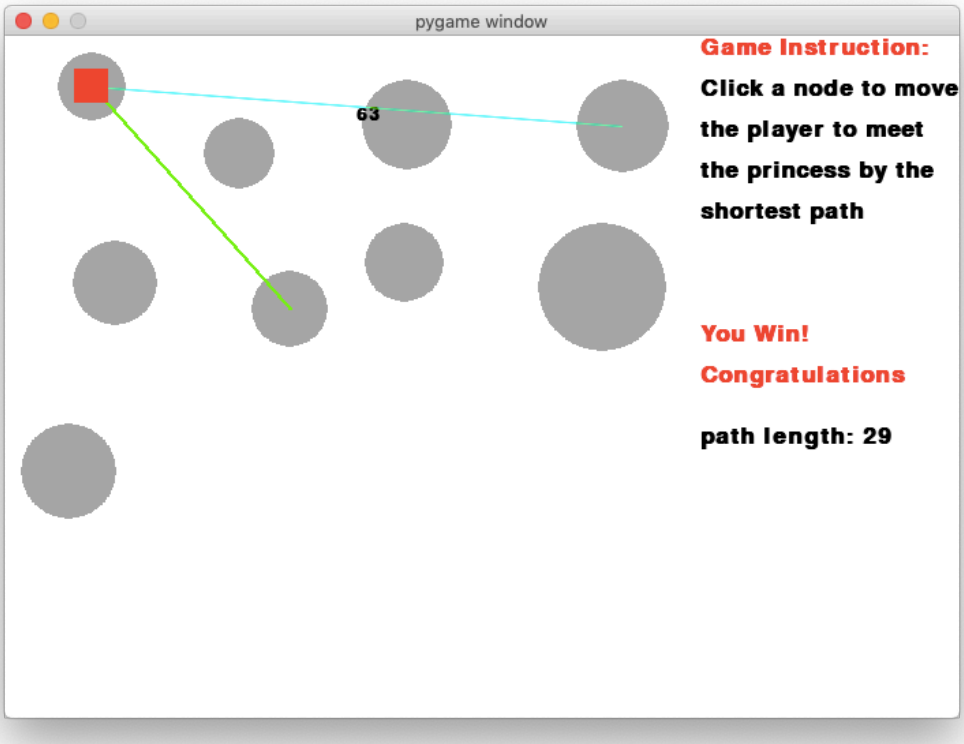
Similar to puzzle1, some of the game information needs to be displayed to help the player understand the game, which is, again, required by the requirement 3 and 6. This includes: game instructions, the correct answer to the puzzle, the length of the path that has already been taken, whether the game is finished, etc. The main method display_info also overrides the method in the parent class Puzzle.

```
def display_info(self):  # display necessary information of the game, such as life, time steps
    # game instruction
    gameInstruction = []
    gameInstruction.append(font.render("Game Instruction:", True, RED))
    gameInstruction.append(font.render("Click a node to move", True, BLACK))
    gameInstruction.append(font.render("the player to meet", True, BLACK))
    gameInstruction.append(font.render("the princess by the", True, BLACK))
    gameInstruction.append(font.render("shortest path", True, BLACK))
    for i in range(len(gameInstruction)):
        screen.blit(gameInstruction[i], [500 + 10, i * 30])

    # game information
    if self.my_player.get_cor() == Puzzle2.princess_cor and self.my_player.get_path() == self.solution:
        screen.blit(font.render("You Win!", True, RED), [screenSize[1] + 10, 200 + 10])
        screen.blit(font.render("Congratulations", True, RED), [screenSize[1] + 10, 200 + 10 + 30])
    elif self.my_player.get_cor() == Puzzle2.princess_cor:
        screen.blit(font.render("Well done!", True, RED), [screenSize[1] + 10, 200 + 10])
        screen.blit(font.render("Try to do it with", True, RED), [screenSize[1] + 10, 200 + 10 + 20])
        screen.blit(font.render("fewer moves", True, RED), [screenSize[1] + 10, 200 + 10 + 20 * 2])

    screen.blit(font.render("path length: " + str(self.my_player.get_path()), True, BLACK), [screenSize[1] + 10, 275 + 10])
    screen.blit(font.render("shortest path: " + str(self.solution), True, BLACK), [screenSize[1] + 10, 275 + 10 + 30])
```

Testing

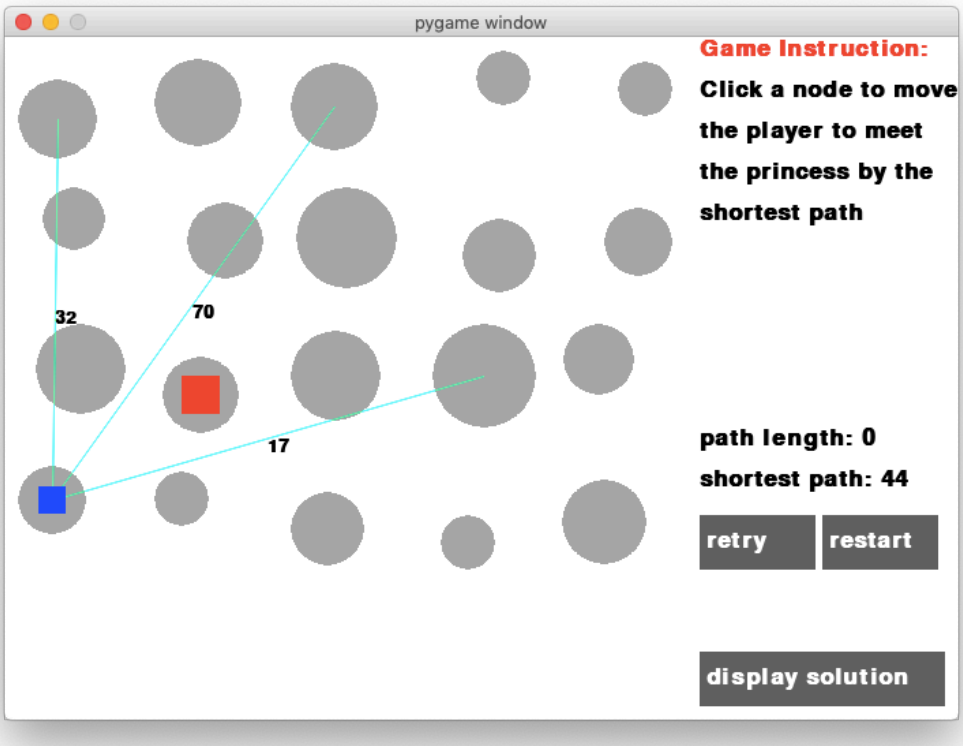
The test shows that the game matches the expectation of test No. 15.



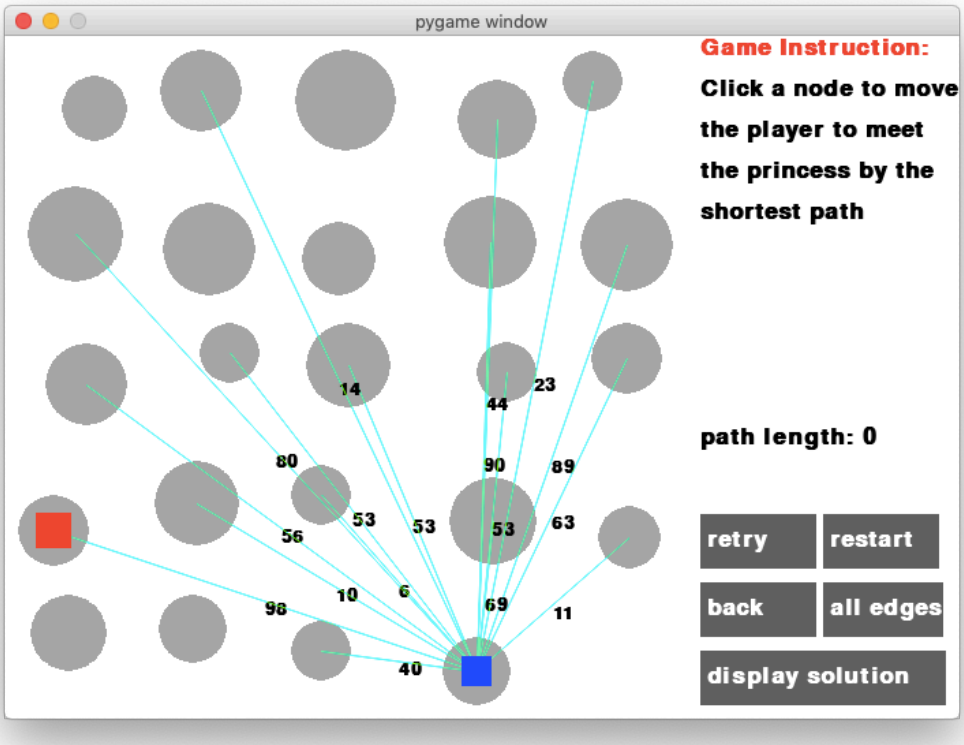


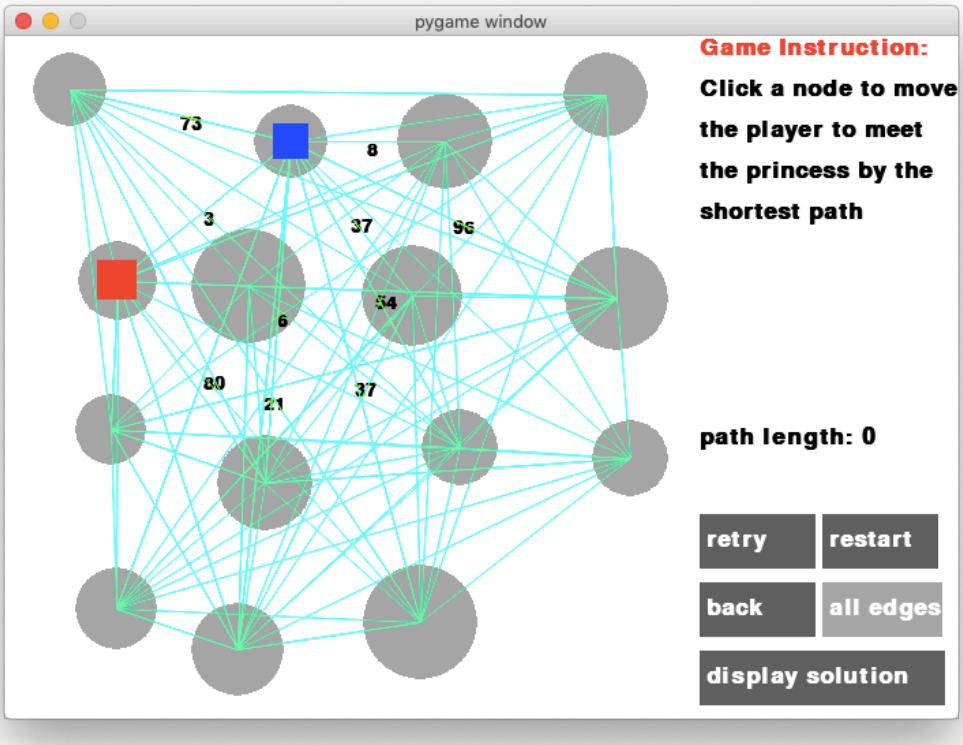
Button functions

As with puzzle1, puzzle 2 will also have three basic buttons available, which are retry, restart and display solution, specified by the same requirements. The implementation of these buttons is similar to the ones in the previous puzzle, and is not included here.



However, while playing the game, it is really easy for a user to make a wrong move and there is currently no way for the player to undo the movement. Furthermore, according to the specification, it is also essential for the game to have a good user experience. It would be really awkward if the user had to start the game again every time he/she makes a wrong move. Therefore, it might be worthwhile to include a back button that allows the user to move one step back. In addition, a button that can display all edges will also be included in case users want to have a look at the overall structure of the graph.





All the buttons in this puzzle are now working properly and therefore the tests 16-20 are passed. In addition, two extra buttons back, and all edges are added to enable better user experience and meets the requirement 12.

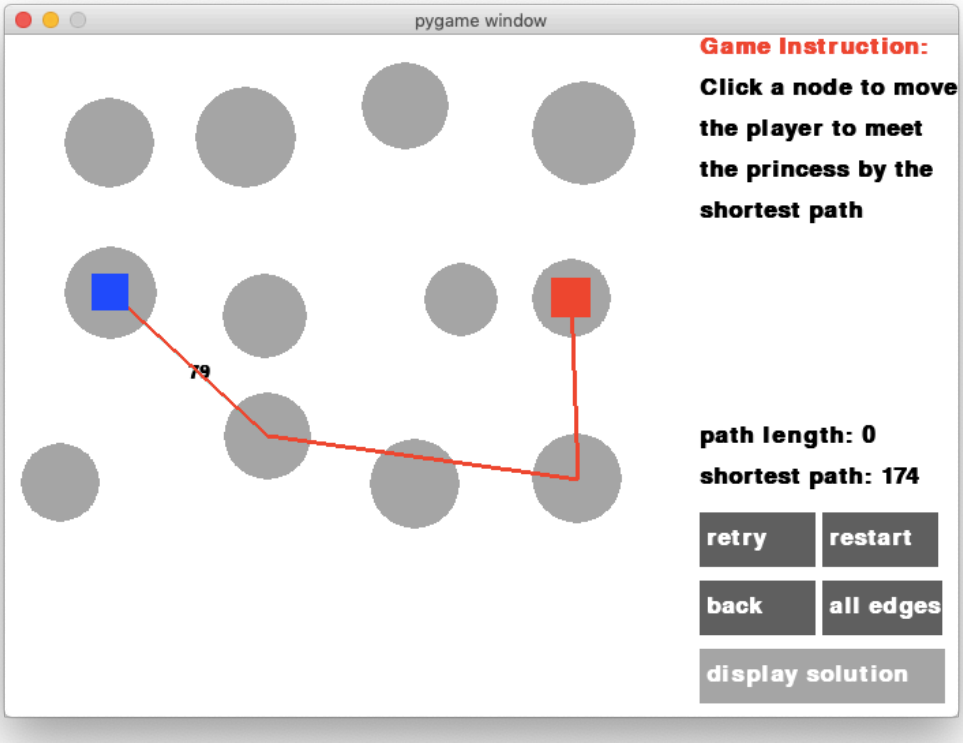
Solution in Dijkstra's

By the review of puzzle1 and requirements 3 and 16, the solution to a puzzle should not only give a numerical answer, but also display it graphically, so that the user can find it easier to trace through the problem rather than struggling with figuring out how the answer is obtained. The answer is found using a famous path-finding algorithm – Dijkstra's. This choice has also been justified in the pseudocode section of the design.

```
def get_solution(self):  # find the optimum solution of a problem
    self.solution_list=[]
    visited=[0]*Puzzle2.node_num
    dist=[INF]*Puzzle2.node_num
    prev=[0]*Puzzle2.node_num
    pq=queue.PriorityQueue()
    nd=copy.deepcopy(Puzzle2.node_list[self.my_player.get_cor()])
    nd.set_weight(0)
    pq.put(nd)
    prev[nd.num]=-1
    dist[nd.num]=0
    while not pq.empty():
        nd=pq.get()
        if visited[nd.num]:
            continue
        if nd.num == Puzzle2.princess_cor:
            self.solution=dist[nd.num]
            father=nd.num
            while True:
                self.solution_list.append(father)
                if prev[father] == -1:
                    break
                father = prev[father]
            self.solution_list.reverse()
            break
        visited[nd.num]=1
        for n in Puzzle2.graph[nd.num]:
            if visited[n.num]:
                continue
            if dist[n.num] > dist[nd.num] + n.weight:
                dist[n.num] = dist[nd.num] + n.weight
                prev[n.num] = nd.num
                pq.put(n)
```

Testing

Again, when the display solution button is pressed, the graphical solution will be displayed on screen in red lines as follows:



This has adopted the improvement from the review of stage1 and matches well with the algorithm test of this puzzle.

Review

Similar to puzzle1, puzzle 2 has been completed with 5 steps, and here are the requirements that have been met with the development of this stage:

- Clear instruction on how to play the game
- All information required is displayed
- Player movement
- Essential functions are implemented and available to the user
- Solutions are straightforward for players to follow

Following on the review from stage 1, some good practice is used as an improvement to stage 1 and to keep the code logically consistent and easy to maintain, which is also part of the requirements that code needs to be as elegant as possible. Improvements include defining puzzle as a super class and each individual puzzle will inherit from the general one; defining game elements as classes, which in the case of puzzle 2 are nodes, helps maintain their positions and other attributes. Problems such as player moving off the centre can be easily avoided with this technique, and this is also to reflect the requirement that the idea of OOP is used properly throughout the game.

There are also some further improvements that can be made in future development, based on the development of this stage.

1. Button functions need to be designed according to the characteristics of the current puzzle. For example, in puzzle2, the game would have been much more difficult for the user to play without the back button as the user would need to make sure that every single movement is correct, or they would have to play the game from the beginning again. Including the back function is therefore a natural thing to do, given the fact that this is game is aimed primarily at learners of some algorithms;
2. It is important that the graphical display of the game on the screen includes all necessary information. This idea is based on the edge display and path highlight in stage2. Again, without highlighting the path that has been taken and display edges in a proper way, this game is much harder to play in terms of its usability rather than difficulty. Therefore, it is vital to consider what to display in the development of next stage so as to make the game as straightforward as possible.

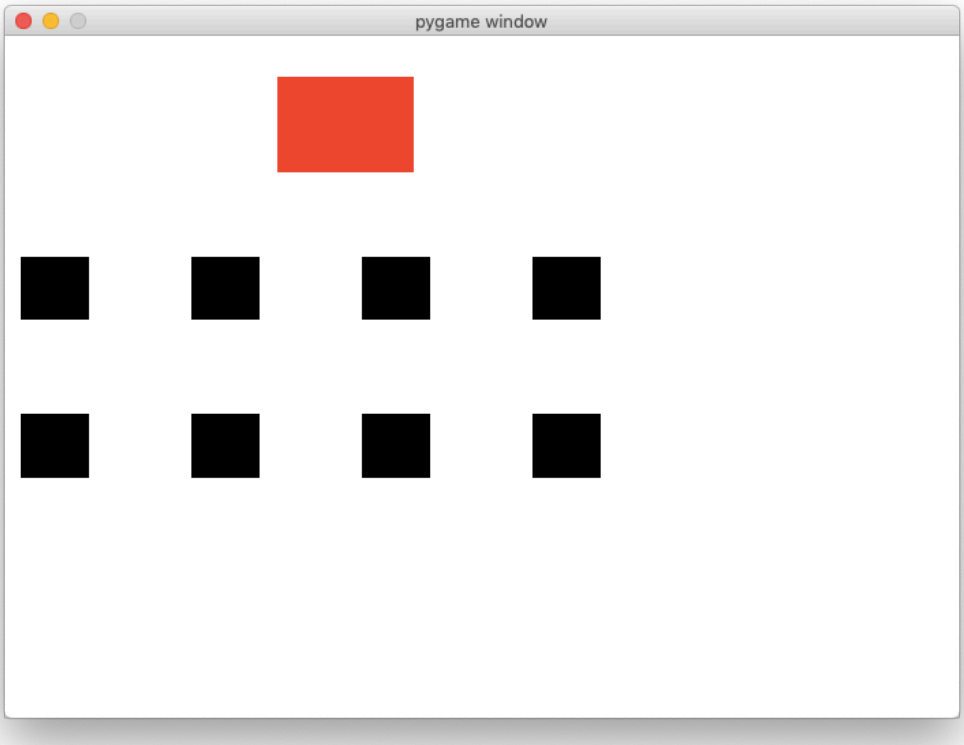
Stage3: puzzle3

This puzzle is to simulate a famous NP problem in computer science – the knapsack problem. Similar to previous two puzzles, the third puzzle is divided into five steps. Here is how it is divided.

Bag and items generation

By requirement No. 5, before the game can start, it is essential to illustrate the problem. Some elements need to be generated and displayed on the screen, i.e. a bag to hold elements (represented by a red box) with particular volume and weight and items of different weights and volume (represented by black boxes). The number of items will be again, generated randomly. Corresponding codes are in the initialise method of the class Puzzle3.

```
def initialise(self):
    Puzzle3.item_num=random.randrange(4, 30)
    self.my_player = Bag([250, 65], [100, 70], RED, random.randint(10, 100))
    all_sprites_group.add(self.my_player)
    num_x = int(math.sqrt(10 / 7 * Puzzle3.item_num)) + 1
    num_y = int(math.sqrt(7 / 10 * Puzzle3.item_num)) + 1
    sep_x = 500/num_x
    sep_y = 350/num_y
    i = 0
    j = 0
    for k in range(Puzzle3.item_num):
        start_x = int((j+0.3)*sep_x)
        end_x = int((j+0.7)*sep_x)
        start_y = 150+int((i + 0.3) * sep_y)
        end_y = 150 + int((i + 0.7) * sep_y)
        item_pos = [start_x,start_y]
        size = [end_x-start_x,end_y-start_y]
        v = random.randrange(5,50)
        w = random.randrange(1,100)
        item = Item(item_pos,size,BLACK,v,w)
        Puzzle3.item_list.append(item)
        all_sprites_group.add(item)
        j += 1
        if j == num_x:
            i += 1
            j %= num_x
```



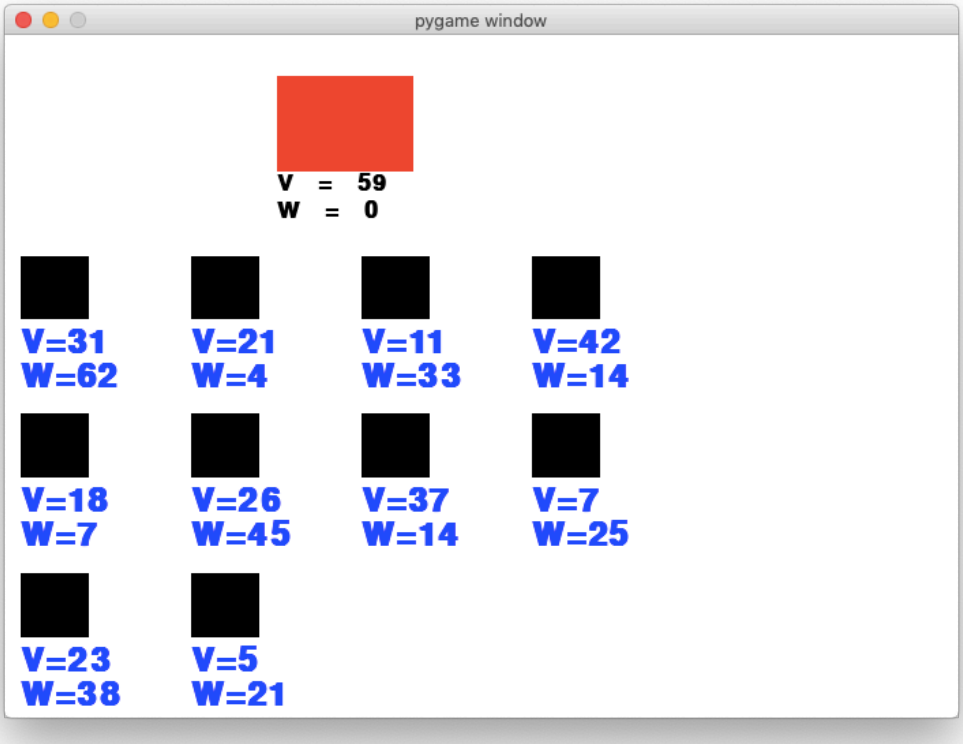
Next, it is essential to display weights and volume of every element displayed on the screen so that the player can see the progress of the game as well as help them decide which items to choose.

```
def display_weight_and_volume(self):
    p = self.my_player.get_pos()
    s = self.my_player.get_size()
    pos = [p[0], p[1]+s[1]]
    font = pg.font.SysFont('Calibri', 25, True, False)
    screen.blit(font.render("V   =   "+str(self.my_player.get_volume()), True, BLACK),
pos)
    screen.blit(font.render("W   =   "+str(self.my_player.get_weight()),True,BLACK),
[pos[0],pos[1]+20])

    for item in Puzzle3.item_list:
        p = item.get_pos()
        s = item.get_size()
        pos = [p[0], p[1]+s[1]+5]
        font = pg.font.SysFont('Calibri', int(s[0]*0.7), True, False)
        screen.blit(font.render("V=" + str(item.get_volume()), True, BLUE), pos)
        screen.blit(font.render("W=" + str(item.get_weight()), True, BLUE), [pos[0],
pos[1] + s[0]*0.5])
```

Testing

The picture below shows that the bag and items are properly displayed, with volume and weight underneath, which is what is expected according to the testing design 23 and 24.

So far, the core elements of this game have been generated, and I can move on to the next step.

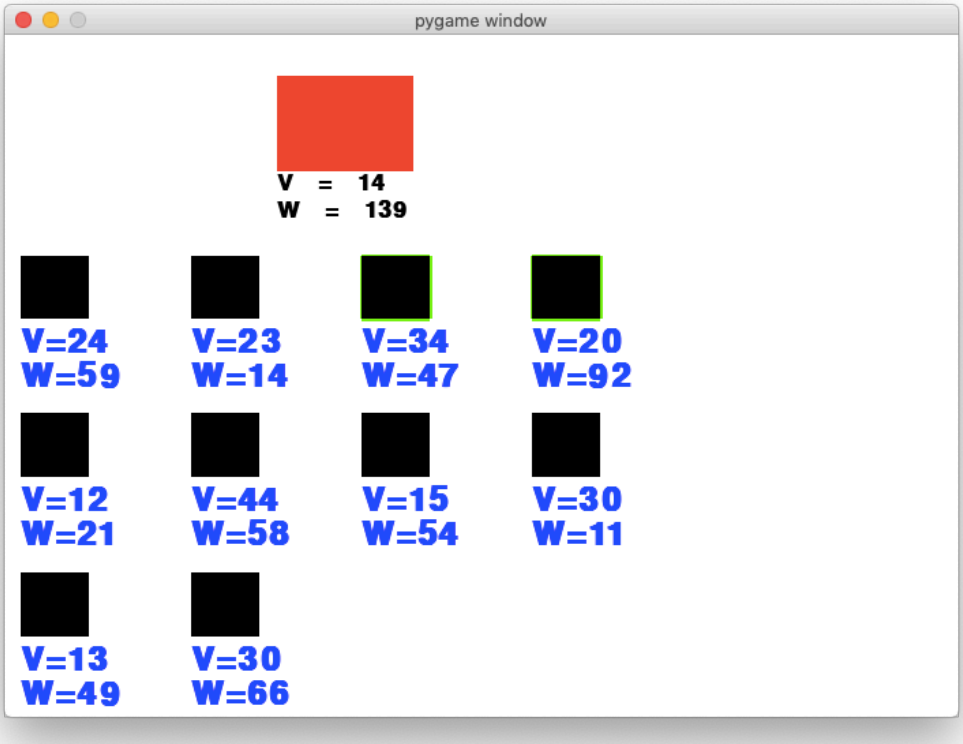


Click to select and deselect

The fundamental move one can make in the game is to choose, therefore, it is important to give the user the ability to choose items in some way as it is required that all essential functions need to be made available to the user. Here I decided to let the user click to choose items that they want, as this is the easiest way in terms of making choices as far as I am concerned. In order to show the user that he/she has chosen some items, those that have been clicked will be highlighted in green. This will be implemented as a method of the class Puzzle 3 and will be run constantly in the main loop as well.

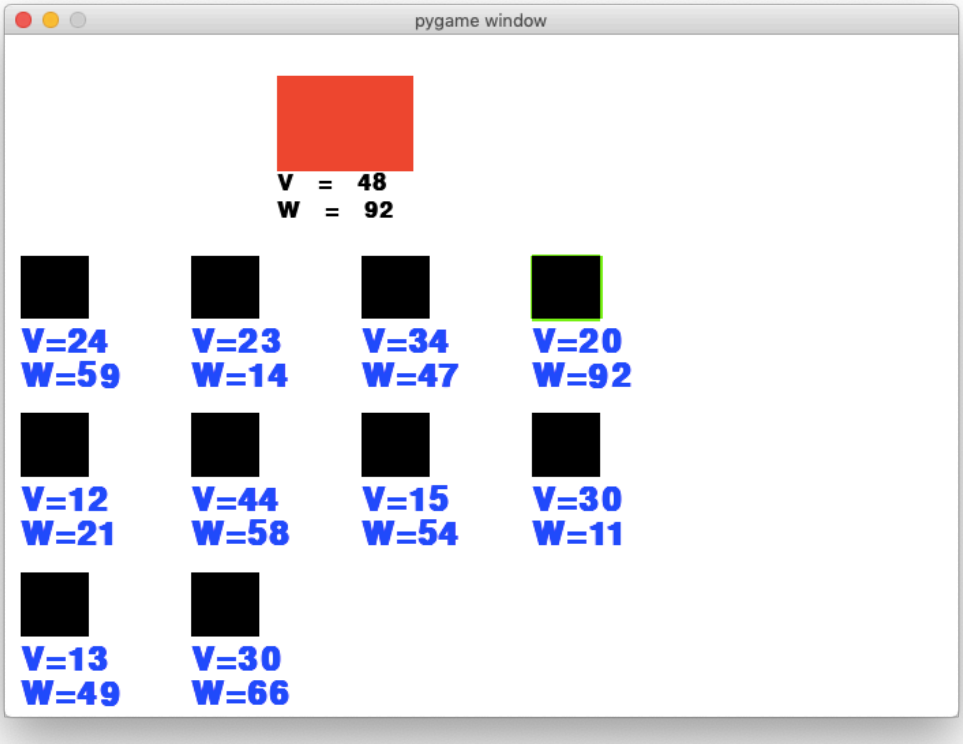
```
def item_highlight(self):
    for item in Puzzle3.item_list:
        added = item.clicked()
        if item.selected:
            item.highlight(GREEN)
        if added == 1:
            self.my_player.weight += item.get_weight()
            self.my_player.volume -= item.get_volume()
        elif added == -1:
            self.my_player.weight -= item.get_weight()
            self.my_player.volume += item.get_volume()
```

Testing



Up to this stage, the user is free to make choices, however, it is very likely that the user wants to undo his/her decision as sometimes a wrong decision could be made but only noticed later on. Similar to the second puzzle, if the user is not able to deselect an item, which means he/she will need to retry the game from the beginning, it can be a serious drawback of the game in terms of its usability. Therefore, both selecting and deselecting are included in the methods of items.

```
def clicked(self):
    if self.mouse_pressed() == 1 and not self.selected:
        self.selected = True
        return 1
    elif self.mouse_pressed() == 2 and self.selected:
        self.selected = False
        return -1
    else:
        return 0
```



With this implementation, the user can now left click to select items and right click to deselect, which are two of the basic functions of this game, and now it has met two of the criteria set in the testing design regrading selecting and deselecting.

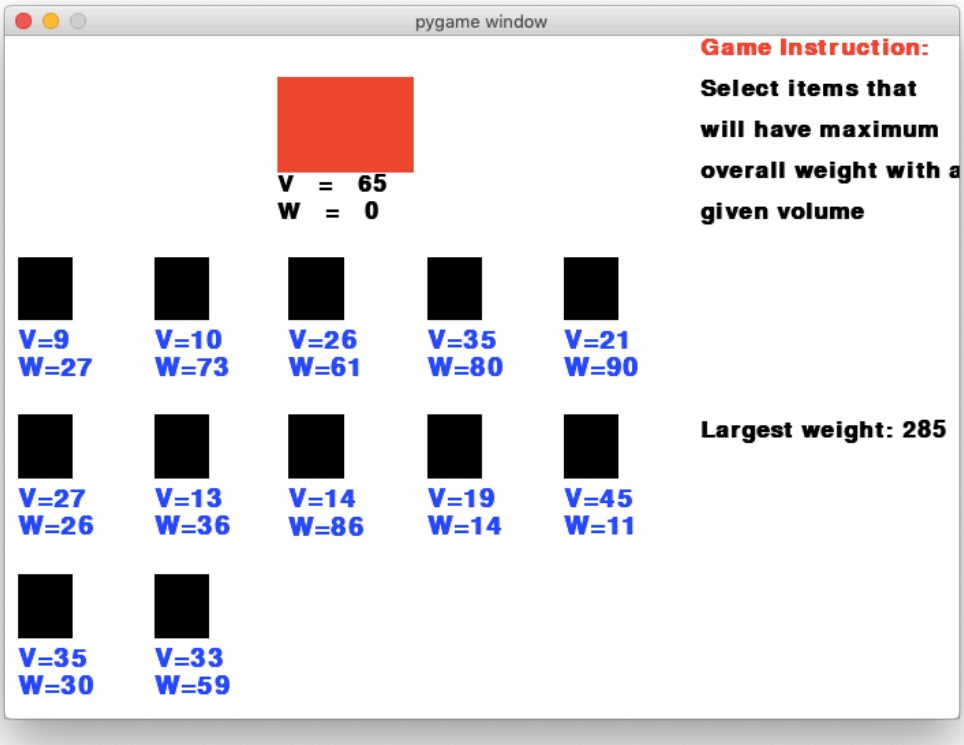
Information display

At this phase, some game information needs to be displayed, these include game instructions, the outcome of the game, etc. The largest weight the player should be aiming for will also be displayed, the numerical value of which will be calculated at the final step. Just like the previous two stages, the code of this part will be implemented as a method of class Puzzle3.

```
def display_info(self):  # display necessary information of the game, such as life, time steps
    # game instruction
    gameInstruction = []
    gameInstruction.append(font.render("Game Instruction:", True, RED))
    gameInstruction.append(font.render("Select items that ", True, BLACK))
    gameInstruction.append(font.render("will have maximum", True, BLACK))
    gameInstruction.append(font.render("overall weight with a", True, BLACK))
    gameInstruction.append(font.render("given volume", True, BLACK))
    for i in range(len(gameInstruction)):
        screen.blit(gameInstruction[i], [500 + 10, i * 30])

    # game information
    if self.my_player.get_weight() == self.solution:
        screen.blit(font.render("You Win!", True, RED), [500 + 10, 200 + 10])
        screen.blit(font.render("Congratulations", True, RED), [500 + 10, 200 + 10 + 30])

    if self.my_player.get_volume() < 0:
        screen.blit(font.render("Whoops! Right lick ", True, RED), [500 + 10, 200 + 10])
        screen.blit(font.render("selected item again", True, RED), [500 + 10, 200 + 10 + 20])
        screen.blit(font.render("to deselect it", True, RED), [500 + 10, 200 + 10 + 20 + 20])

    screen.blit(font.render("Largest weight: " + str(self.solution), True, BLACK), [500 + 10, 280])
```

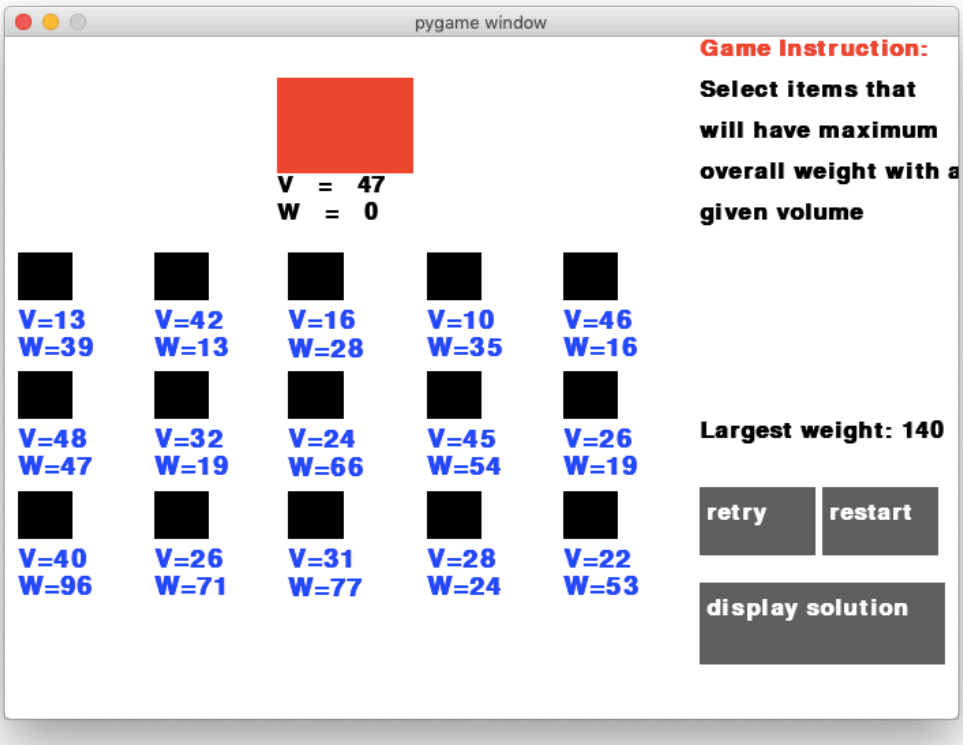
Testing

The game is run and tested, one can see the effect above as expected and hence it has successfully passed the test No. 25.



Button functions

A common feature to all three puzzles is button, in the case of puzzle3, there is not any particular important functionalities to be implemented as buttons, as one of the most important functions is achieved by right clicking an item. Therefore, buttons in this puzzle will be the same as puzzle1 according to the same requirement, so the code is not included here.



The function of a display solution button will, however, be dependent on the solution to this puzzle and will therefore not work at this step. I will finish the function of this button when I have finished the next step, which is to calculate the answer using dynamic programming.

Solution in DP

Finally, the solution of the puzzle will need to be calculated and displayed in the game information section on the right-hand side as shown at the information display stage. Here dynamic programming is used to calculate the answer as it is both quick and easy to implement.

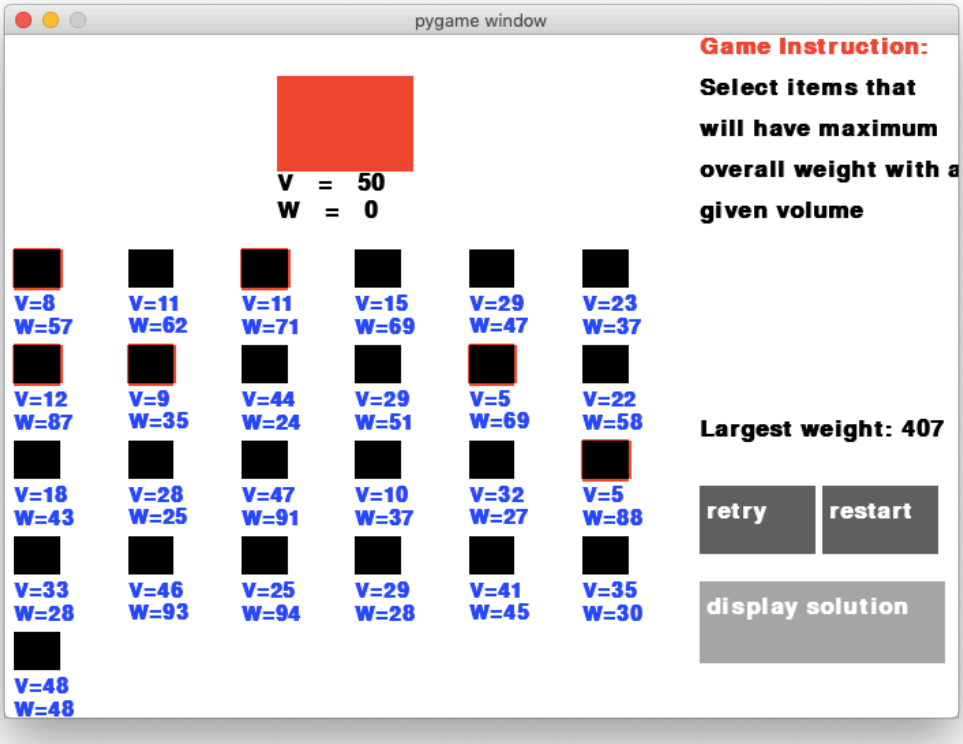
Furthermore, according to the review of stages one and two, not only the numerical answer but also the graphical solution will need to be displayed on the screen to help users understand how the algorithm works, which is also a requirement that solutions are straightforward for players to follow. Here, an additional array is used to store items that are selected so that they can be displayed on the screen.

All of these codes are included as a method of the class Puzzle three.

```
def get_solution(self):  # find the optimum solution of a problem
    w = [0]*(Puzzle3.item_num + 1)
    v = [0]*(Puzzle3.item_num + 1)
    for i in range(1, Puzzle3.item_num + 1):
        w[i] = Puzzle3.item_list[i-1].get_weight() # i-1 is because index starts from 0
in the item_list
        v[i] = Puzzle3.item_list[i-1].get_volume()

    dp = [[0] * (self.my_player.volume+1) for i in range(Puzzle3.item_num + 1)]
    choices = [[[]] * (self.my_player.volume + 1) for i in range(Puzzle3.item_num + 1)]
    # store how items are selected for dp[i][j]
    for i in range(1, Puzzle3.item_num + 1):
        for j in range(1,self.my_player.get_volume()+1):
            dp[i][j] = dp[i-1][j]
            choices[i][j] = choices[i-1][j]
            if j >= v[i] and dp[i-1][j-v[i]]+w[i] > dp[i-1][j]:
                dp[i][j] = dp[i-1][j-v[i]]+w[i]
                choices[i][j] = choices[i-1][j-v[i]] + [i]

    self.solution = dp[Puzzle3.item_num][self.my_player.get_volume()]
    self. solution_list = choices[Puzzle3.item_num][self.my_player.get_volume()]
```

Testing

Once the display solution button is pressed, the solution will be displayed on the screen, with items need to be selected highlighted in red. This test does pass the 19th and 33rd test as listed above.



Review

During this stage, the following criteria have been met:

- Clear instruction on how to play the game
- All information required is displayed
- Essential functions are implemented and available to the user
- Solutions are straightforward for players to follow

It has also been tested against the testing design, and all eleven tests designed for this puzzle are now, with some modifications and iterative development, passed.

The main part of this game has now been completed with three stages, in other words, all the key elements are now working, leaving only one final thing to be completed: the puzzle selection user interface, and it will be dealt with in the final stage.

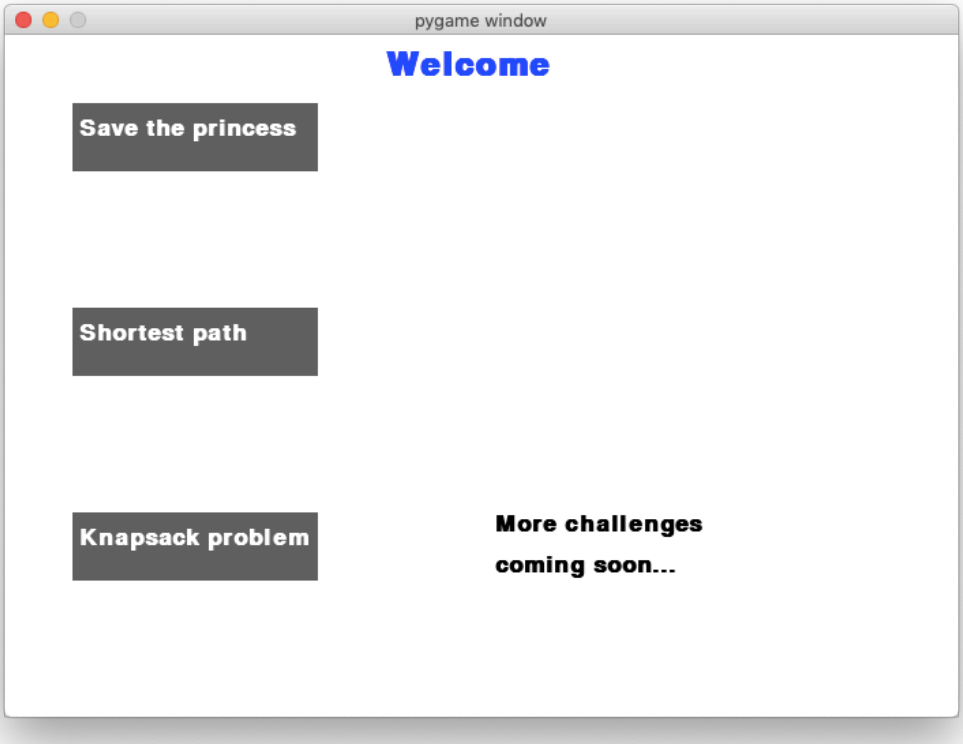
Final stage: puzzle selection

Compared with the other three stages, this stage is relatively straightforward and does not need much careful break-down. The basic idea is that there will be three buttons displayed, each leading to a specific puzzle. Once one of the buttons is pressed, the program will respond and go to the corresponding puzzle. However, it is very important that this selection UI is available to the user otherwise there will be no way to choose among three puzzles from the user point of view, which will fail to meet the 12th requirement. The puzzle selection will also be defined as a class which inherits the Puzzle class.

```
class PuzzleSelection(Puzzle):
    def __init__(self):
        super().__init__()
        self.my_player = Element([0, 0], [0, 0], BLACK)
        self.puzzle1 = Button([50, 50], [180, 50], GREY, "Save the princess")
        self.button_list.append(self.puzzle1)
        self.puzzle2 = Button([50, 200], [180, 50], GREY, "Shortest path")
        self.button_list.append(self.puzzle2)
        self.puzzle3 = Button([50, 350], [180, 50], GREY, "Knapsack problem")
        self.button_list.append(self.puzzle3)
    def display_info(self):
        font1 = pg.font.SysFont('Calibri', 35, True, False)
        screen.blit(font1.render("Welcome", True, BLUE),[280,10])
        screen.blit(font.render("More challenges", True, BLACK),[360,350])
        screen.blit(font.render("coming soon...",True,BLACK),[360,380])
    def button_function(self):
        global curPuzzle
        if self.puzzle1.is_pressed():
            curPuzzle = Puzzle1()
            curPuzzle.pre_update()
        elif self.puzzle2.is_pressed():
            curPuzzle = Puzzle2()
            curPuzzle.pre_update()
        elif self.puzzle3.is_pressed():
            curPuzzle = Puzzle3()
            curPuzzle.pre_update()

        for b in self.button_list:
            b.update()

    def update(self):
        self.button_function()
        self.display_info()
```

Testing

Here is how it looks when the program is run and tested.



Review

This is the last stage of the development cycle and all functions have now been completed. All of the following success criteria are satisfied:

- Clear instruction on how to play the game
- Player movement
- All information required is displayed
- Game running correctly and logically
- Problem illustration as straightforward as possible
- Essential functions are implemented and available to the user
- The user needs to think strategically in order to pass the game
- Principles behind games are logically strict
- Puzzles are chosen based on how classic and how often they are used to solve problems
- Idea of OO is used properly throughout
- Puzzles are carefully selected so that they are neither too obscure nor too basic
- Puzzles are designed to deepen the understanding of algorithms rather than broaden the knowledge

Evaluation

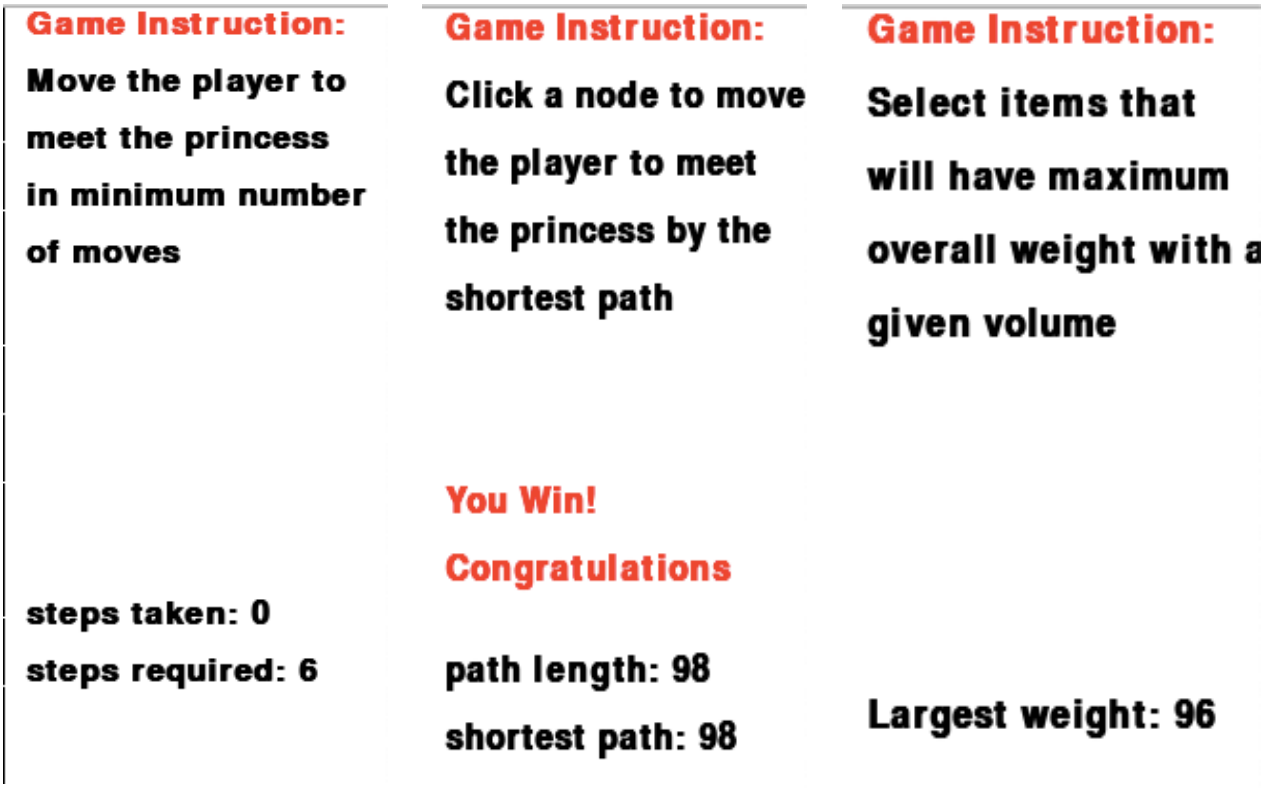
Now it is time to go back and review the final product and compare it with the requirements produced at the analysis stage. As the requirements are divided into three categories, I will evaluate my program in this order.

Success criteria (must)

All key requirements labelled with a must are the success criteria of this game and almost all of them are met, and the evidence of which is provided by carrying out some post development testing as shown below.

1. Clear instruction on how to play the game

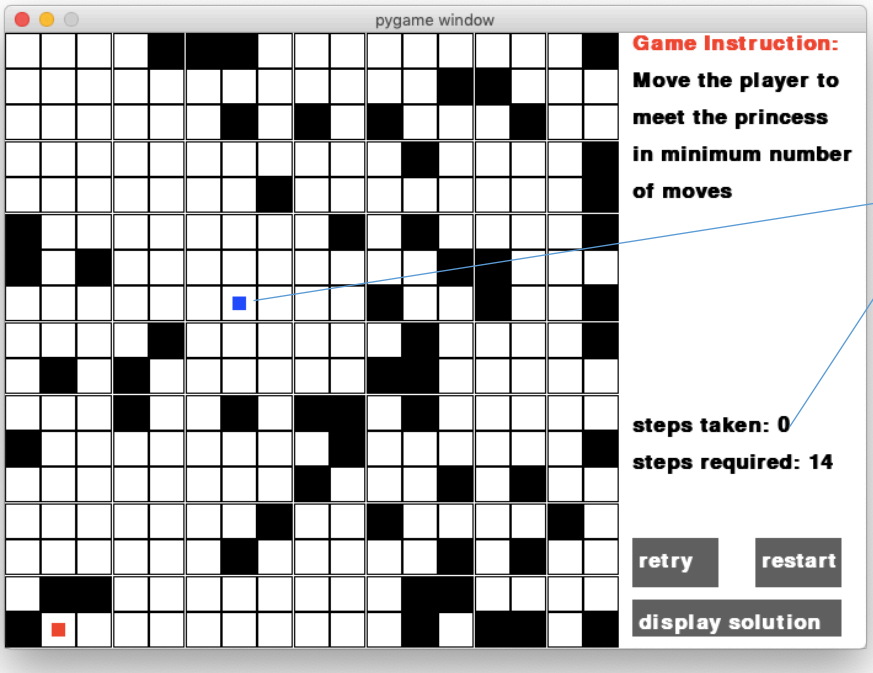
This is to enable the user to start the game smoothly and it is also essential that the user is clear about how to play the game, otherwise it is pointless to develop the game.



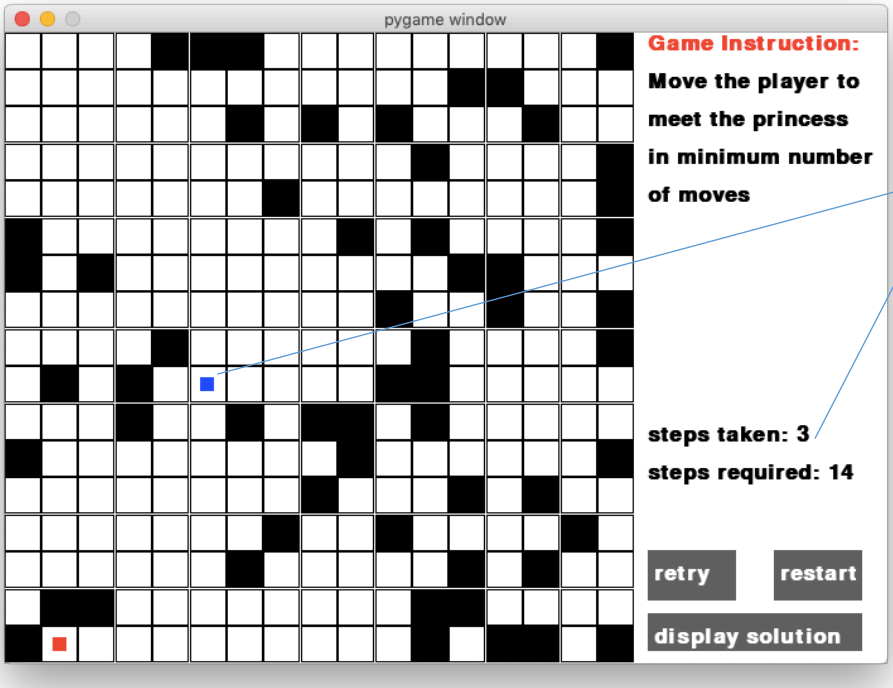
As one can see there is a dedicated area in each of the puzzle to give the user game instructions and other essential information, such as the outcome of the game, so that the user can have a decent understanding of the game after a few trials. This criterion has been met.

2. Player movement

This is another key function of the game – the ability to move. A better and more general way of describing this requirement is that the game can give necessary response to the user input. A series of screenshots below show how this is achieved in each of the puzzles.

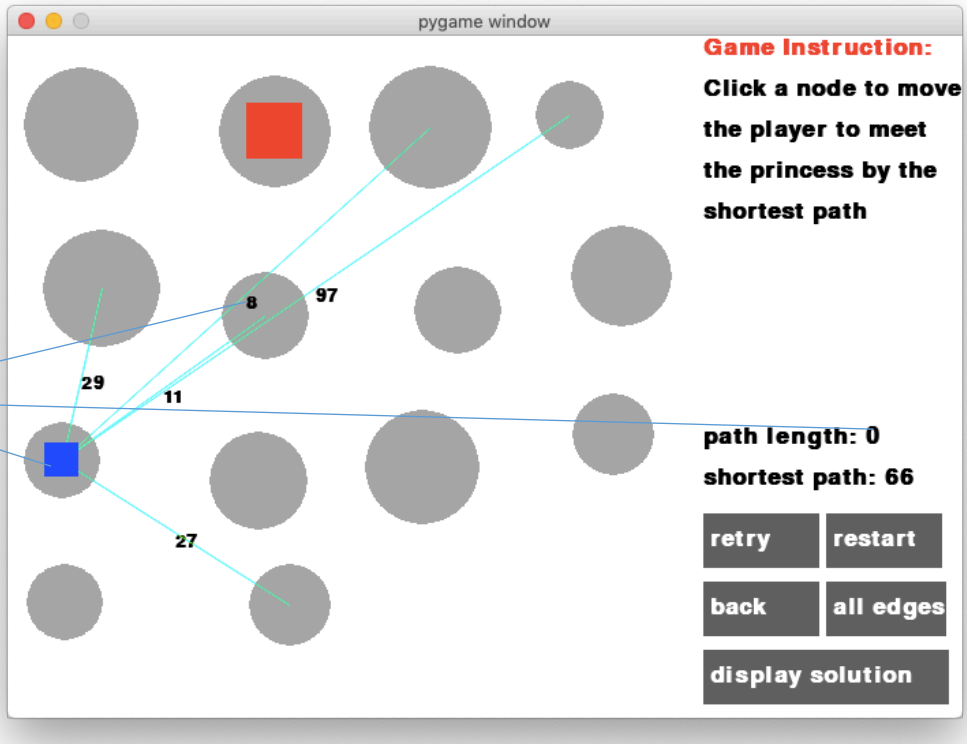


Initial position and the value is zero.

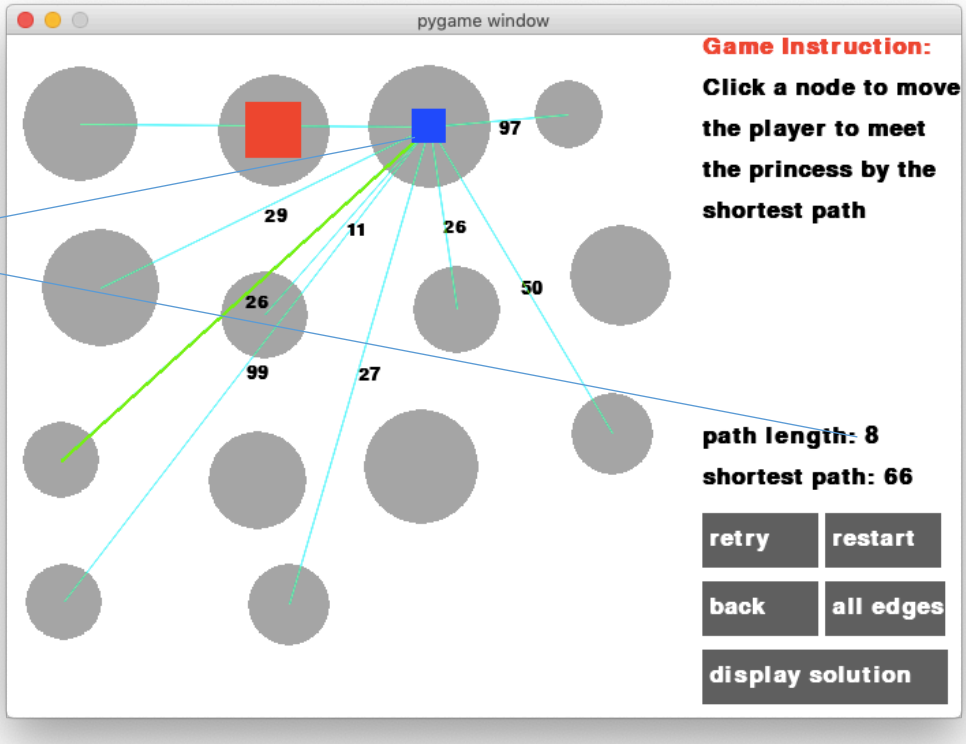


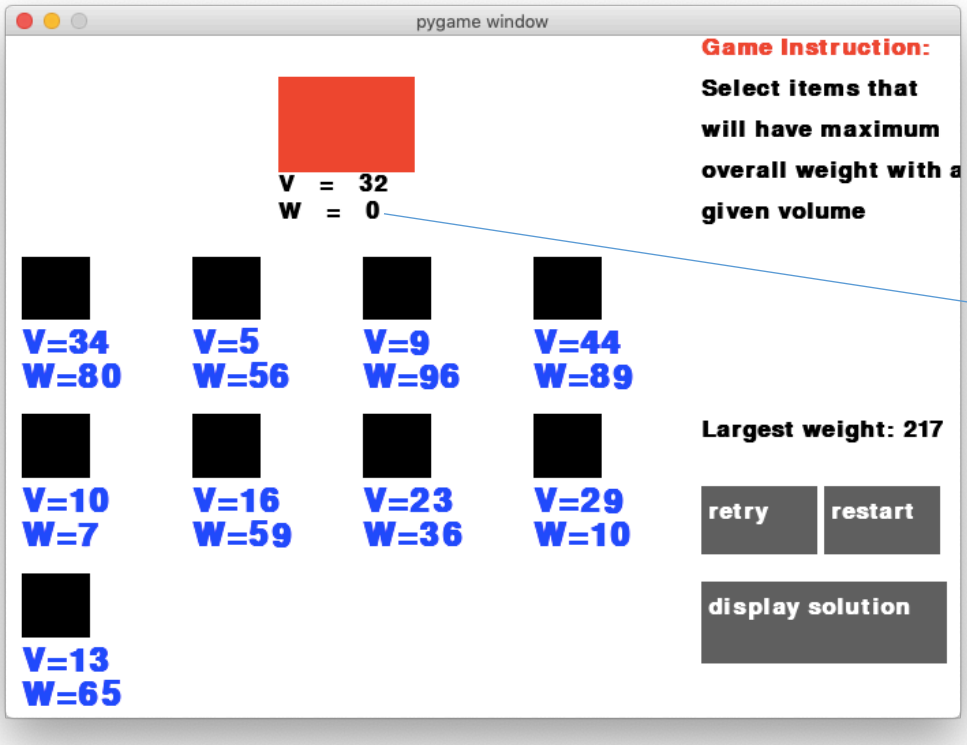
The player has moved three steps and the value becomes three.

The player is now at its initial position, and suppose that it will follow the edge of weight 8.

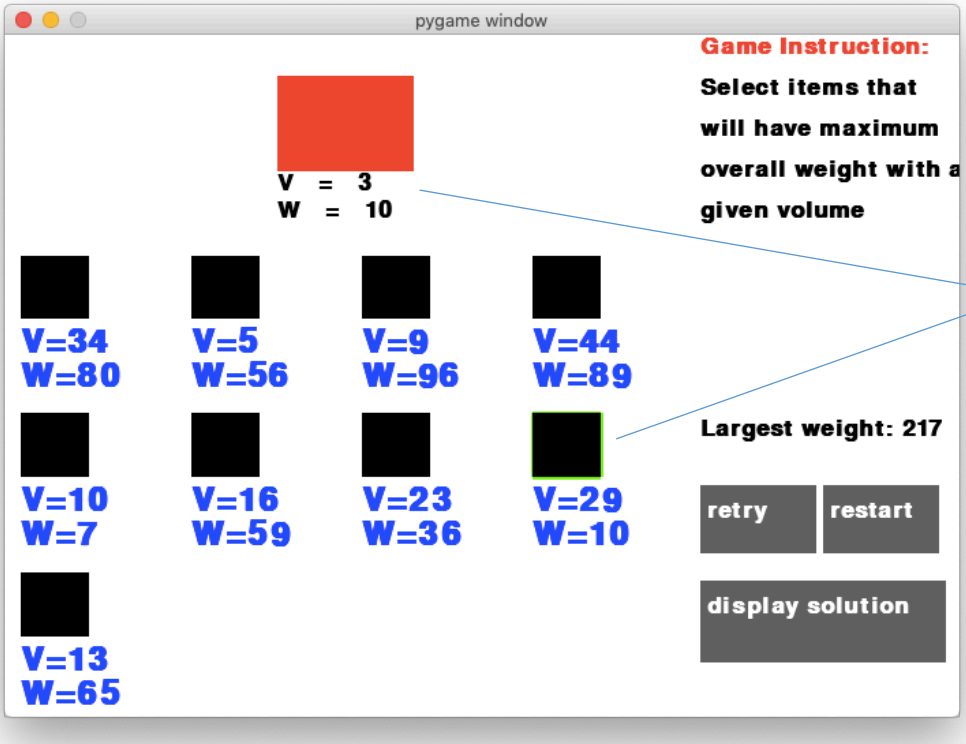


Now the player is in a new node and the path length has been updated.





The weight of the bag is zero at start.

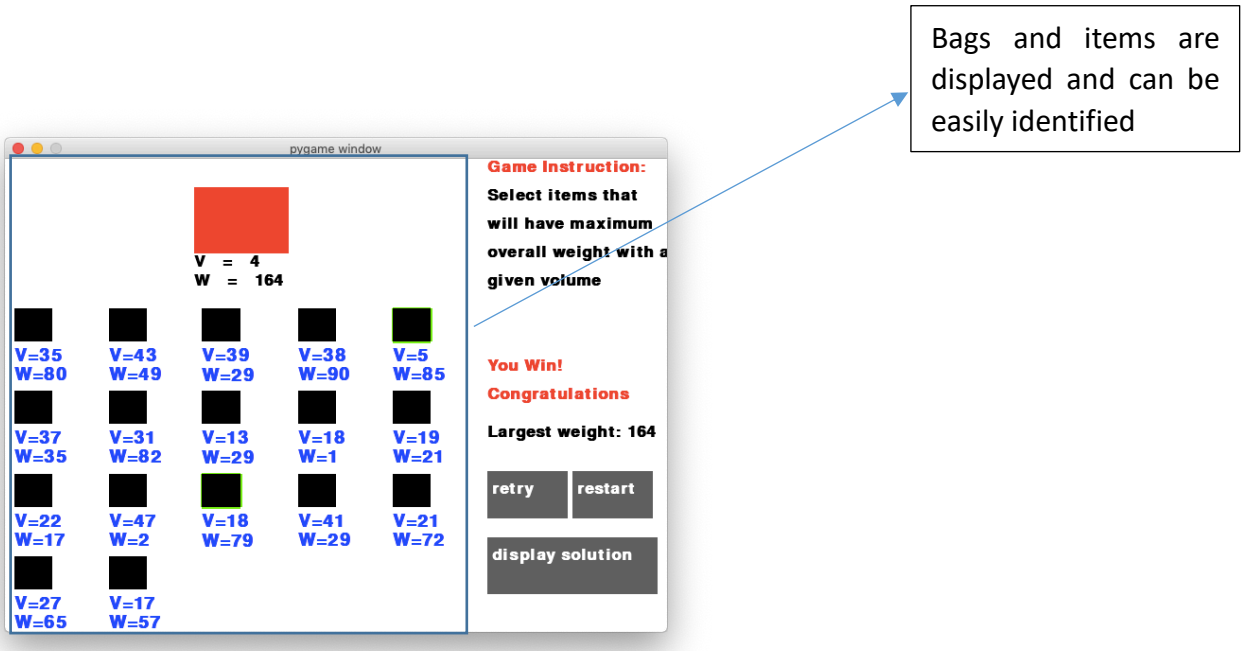


This item is now selected and the volume and the weight of the bag are changed.

In all three puzzles, this criterion has been achieved – moving with arrow keys in the first puzzle and click to move and select in the second and third puzzle. In other words, all three puzzles will give response to the user inputs as expected and therefore this requirement is met.

3.All information required is displayed

Similar to the first requirement, this requirement is met by displaying the game properly in each of the puzzle without any important information missing. For example, in the third puzzle:

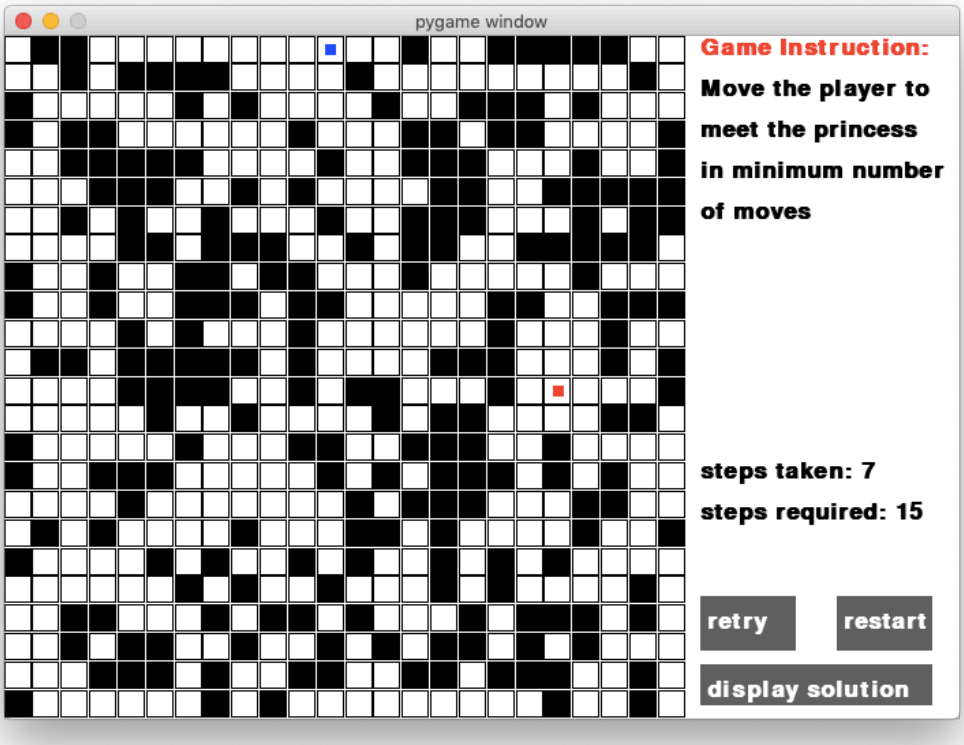


4.Game running correctly and logically

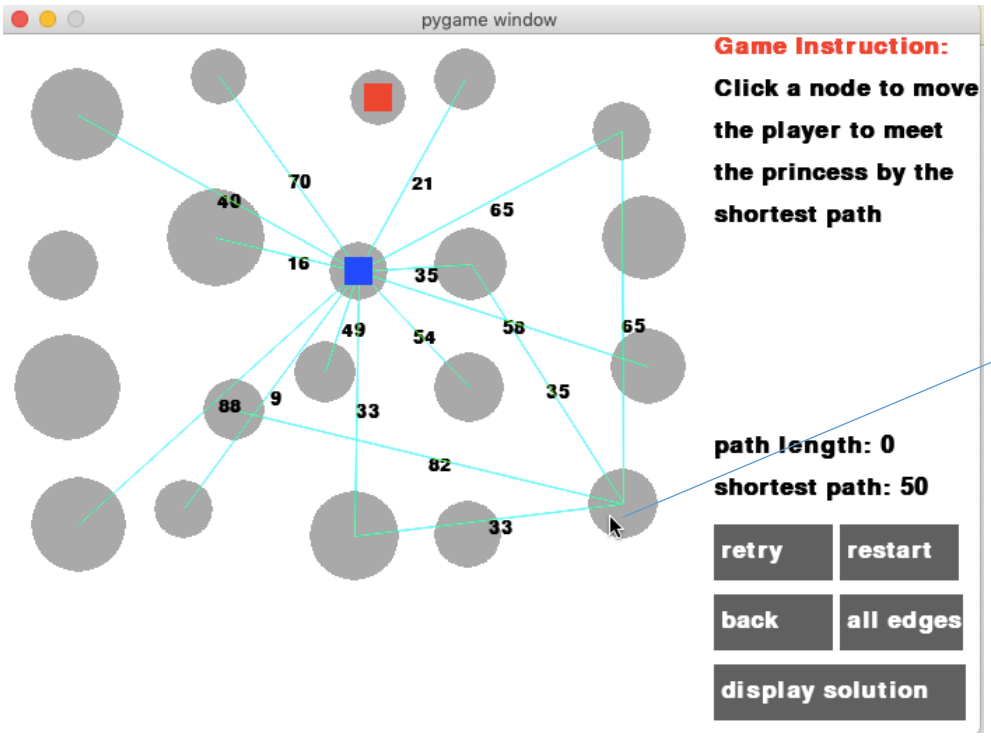
It is important to make sure that the logic of the game is correct, and some simple tests are carried out on each of the puzzles.

For the first puzzle, the movement of the player is essential, therefore I did the test to see what happens if one continues to move the player when it is next to the edge of the map, it turns out that the player will not move as required.

For example, in the screen below, the player is at the upper boundary of the maze, and it will not move any further even if the up-arrow key is pressed.



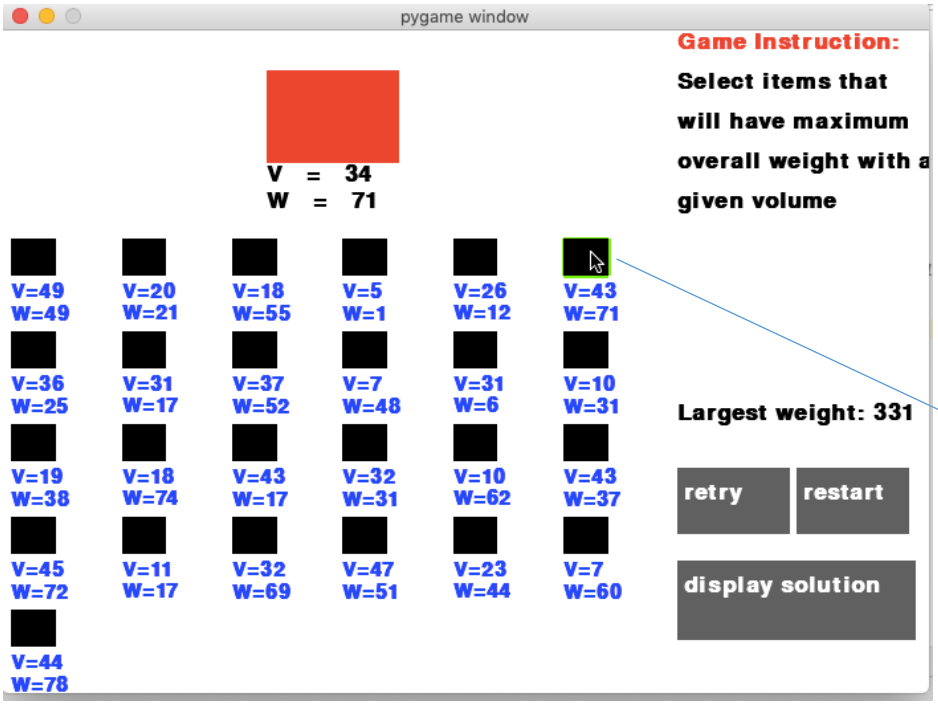
The second puzzle involves moving across different nodes, a simple test would be to see if the player will move between two unconnected nodes by clicking a node that is not connected to the one that the player is currently on. And the result is also as expected.



When clicking on a node that is not connected, the player will not move.

The last puzzle is about selection of items, so I tried to select same items multiple times and the expected outcome is that the weight of the bag stays the same and it turns out to be so.

As one can see that clicking on the same item does not make a difference to the volume and weight of the bag.



Clicking on the same item multiple times does not make a difference.

It is therefore reasonable to say that the game can run correctly and logically by the tests that have been done so far.

19.Puzzles are chosen based on how classic and how often they are used to solve problems

It is not easy to tell how classic an algorithm is, but Breadth-first search, Dijkstra's and dynamic programming are all widely used and considered relatively classic.

20.Idea of OO is used properly throughout

This requirement can be tested by looking at the code. Furthermore, throughout the development stages, there have been some reviews on the use of classes to handle the relationship between puzzles. For example, although each puzzle seems to be very independent of each other, they are all defined as a class and all inherit from the same super class which acts like a virtual class. Similarly, all elements in the game, such as the player in the first two puzzles and the bag in the third one, inherit the same super class Element.

Desirable features (should and could)

Other than the success criteria, there are some other requirements that are not imperative but would be nice to have. They are summarised below.

6.Having seen a similar problem is helpful but not essential to be able to play

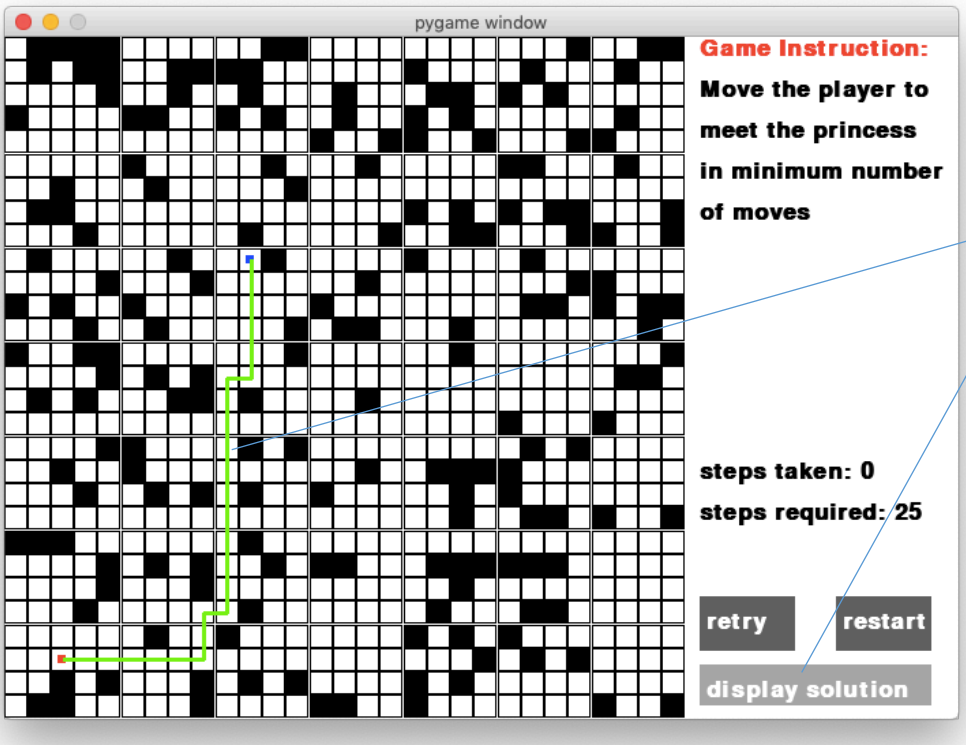
This requirement has also been met, and in fact, it is strongly related to two of the success criteria. As long as the game has clear instruction and all the information required to play the game is available, it would not be too hard for one to understand how to play the game even without much prior experience.

10. Game can run smoothly and provides good user experience

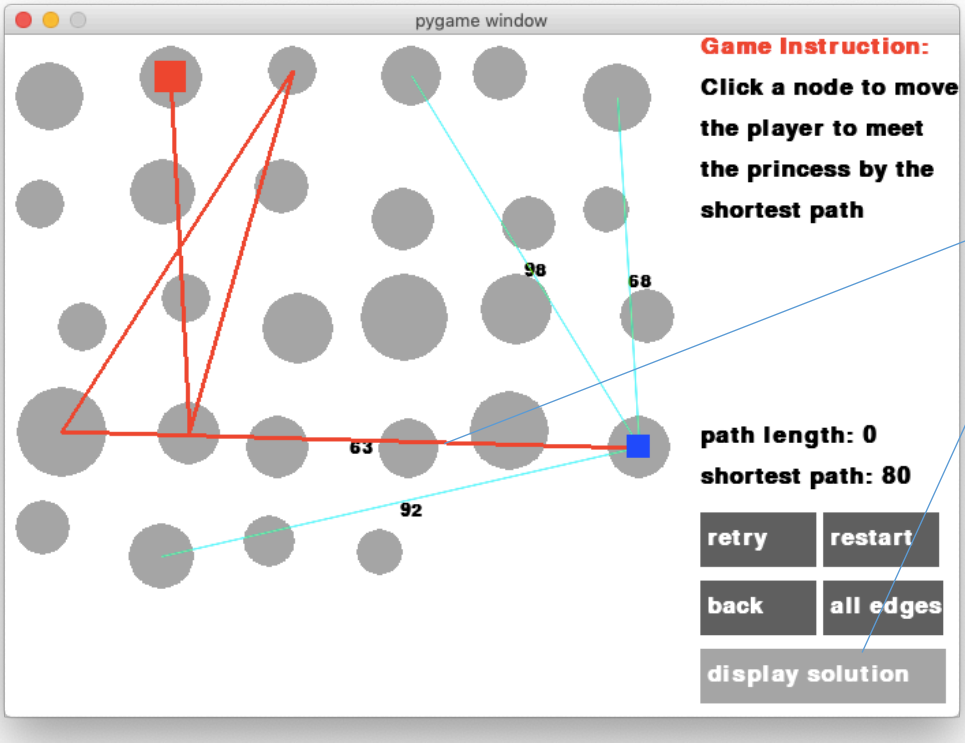
This experience of playing this game is relatively smooth as all the basic functions are available but not entirely enjoyable since there are no transition animations for example, in the game while the user is choosing. Therefore, this requirement is partially met.

16.Solutions are straightforward for players to follow

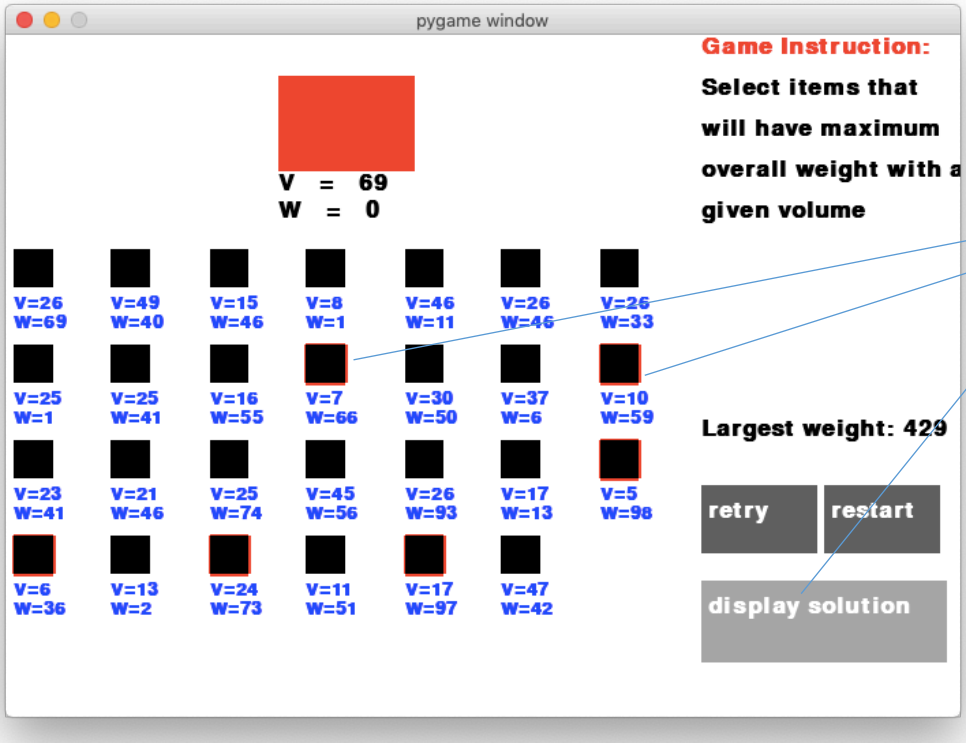
This requirement is met as graphical solutions in addition to numerical answers are provided in all three puzzles so that the user can easily see how to actually obtain the correct answer. By simply pressing the display solution button, the solution will be displayed on the screen.



A green path is displayed when the button is held.



Similarly, a path will be highlighted when the button is pressed.



Items need to be selected will be shown.

Stakeholders and robustness of the program

From the stakeholders' point of view, they might input something unexpected that might crush the game. Therefore, it is important that some destructive testing is done on the game. The following tests have been done.

Test	output
Puzzle 1	
Keys other than arrow keys are pressed	No response
Arrow keys being pressed at the same time	The player will move accordingly, and number of steps increases as expected
Puzzle 2	
Keys are pressed	No response
Click a node many times	No response
Puzzle 3	
Click on the bag	No response
Select all items	Game run as normal
Right click an item many times	No response

Checklist of tests and requirements

Below is a checklist of most of the requirements and to what extent they have been met. They are split into three categories: must, should and could as designed. Some of the requirements do not have a corresponding test number as they are not easily testable, or they are similar to one of the requirements that have already been tested.

Requirement No.	Test No.	Met?	Where it has been tested
Must			
1	3,4,15,25,26	Fully	p.95
2	2,11	Fully	p.96-98
3	3,4,15,25,26	Fully	p.100
4	8,14,28	Fully	p.101
5	1,11,24	Fully	p.54,69,81-83
12	5,6,7,16,17,18,19,20,21,29,39,31,32,33	Fully	p.113-116
13	6,18,30	Fully	p.113
17	10,23,35	Fully	Manually
19	N/A	Partially	p.104
20	N/A	Fully	p.104
23	N/A	Partially	N/A
24	N/A	Fully	N/A
Should			
6	3,4,15,25,26	Fully	p.106
8	N/A	Not	N/A
10	N/A	Partially	p.106
11	N/A	Fully	p.109
15	N/A	Partially	p.107
16	7,18,29	Fully	p.107
Could (most of these are not easily testable)			
7	N/A	Not	N/A
9	N/A	Not	N/A
14	N/A	Partially	N/A
18	N/A	Not	N/A
21	N/A	Partially	p.122
22	N/A	Partially	p.122
25	N/A	Not	N/A

Future development

In terms of future development, there are some requirements that have not yet been met but is worth considering in the future.

- Challenges are offered in various levels of difficulty

There are some other varieties of these puzzles where similar techniques can be used but requires some extra bit of thinking. It might be worthwhile to include the level system so that the user can apply what they have learnt from level1 to level2 which are similar problems but takes one step further.

For example, the first puzzle, saving the princess, is one of the most straightforward applications of BFS, there are lots of variations available, such as adding enemies which will take extra time for the player to fight with the enemy. In this case, some modifications of the original searching algorithms are needed, which can really stretch the user and deepen their understanding of what they have learnt from the simplest version of the algorithm.

The second puzzle, finding the shortest path, may also be improved by including edges of negative weights. Although this involves different algorithms to solve it as Dijkstra's does not apply to graphs with negative weights, this does require the user to think more about the idea that the Dijkstra's is greedy and why being greedy will fail to find the shortest path on a negatively weighted graph.

- Some illustrations of characters may be included

Rather than squares, some characters may be included to make the game more fun. For example, the first puzzle is called save the princess, so the blue box can be replaced by a picture of knight and the red box a picture of princess.

- Extra guidance provided if a user does not understand the problem

Although effort is put into making sure this game is relatively easy to play and very user-friendly, some people might still be confused as to what the goal of the puzzle is and how to play the game. In this case, some extra guidance can be provided by either make the window larger, as currently there is no space for more text, or by using some sort of pop-up window. For instance, a question mark can be placed on the top-right corner of the game window, and when the user clicks on it, it will show more detailed instructions and help on how to play the game. A question mark below may be used.

- Stories are interesting to most players

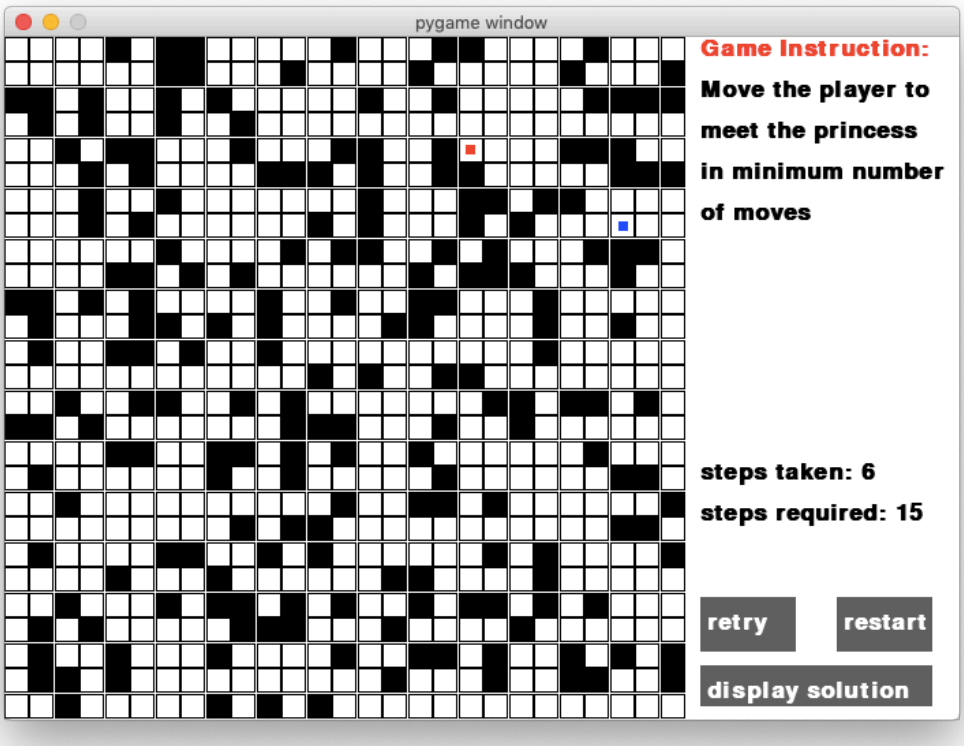
Rather than directly giving instructions to users, which was what I did in all three puzzles, some stories may be given instead that tell the user some background information of the game. For example, in the first puzzle, why and where is the princess trapped, who is going to be save her, etc. This will not only add more fun to the game but also requires the user to think abstractly and keep only the important information needed while they are solving the problem.

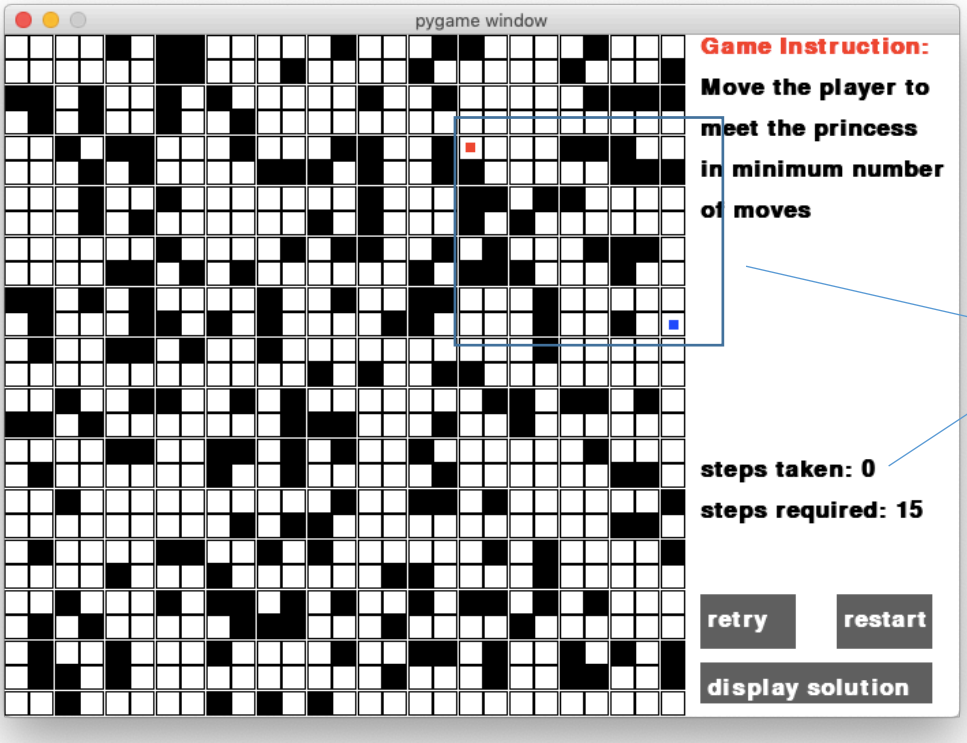
Usability features

There are also some usability features involved in this game in order to give users a smoother experience. One of them is to implement all functions that are necessary. This is primarily achieved by buttons.

There are three buttons that are common to all three puzzles – retry, restart and display solution. These three buttons will give three basic functions, i.e. try the game again with the same or different settings and look at the solution.

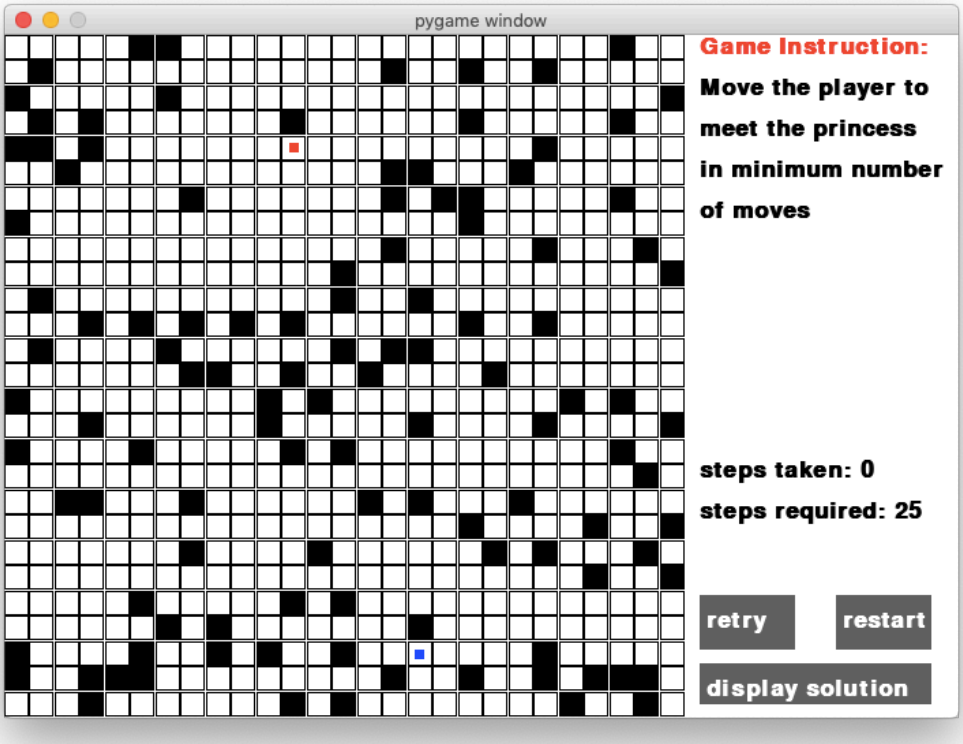
For example, in the first puzzle, saving the princess. If the user wants to go back to the original position at this stage, he/she can simply press the retry button.



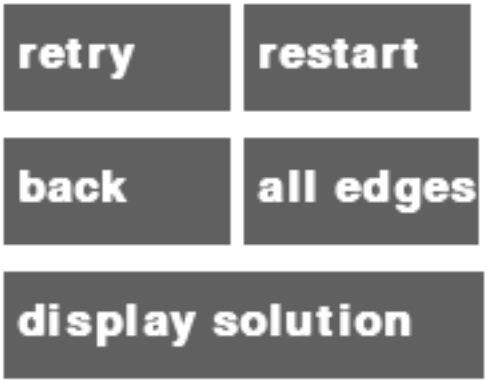


The position and the step have been reset.

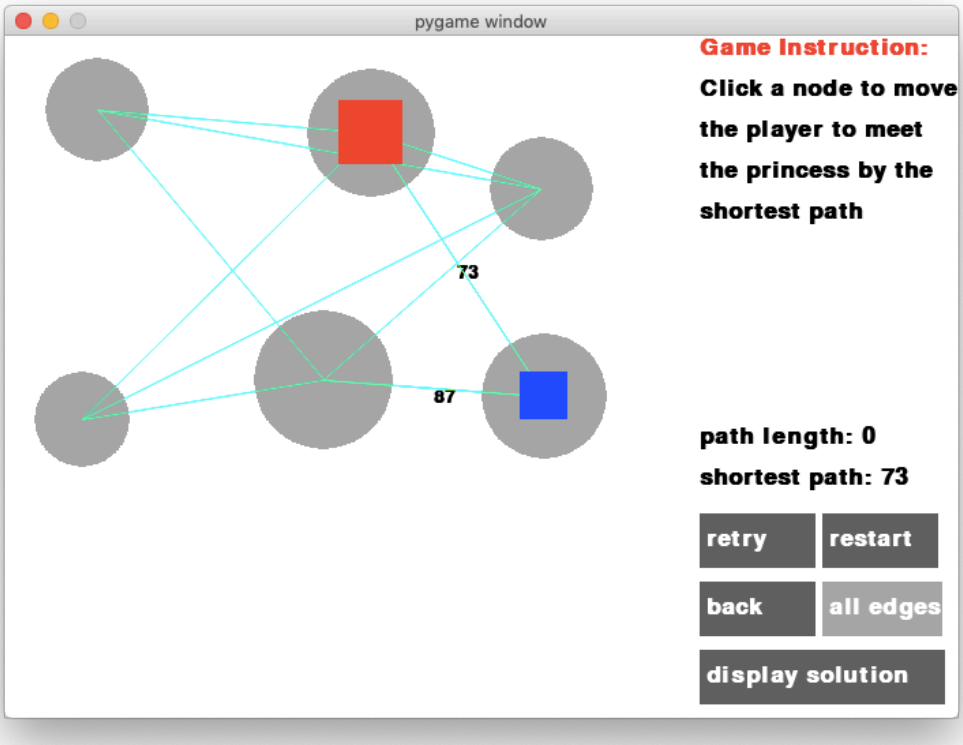
And the restart button gives a whole new maze.



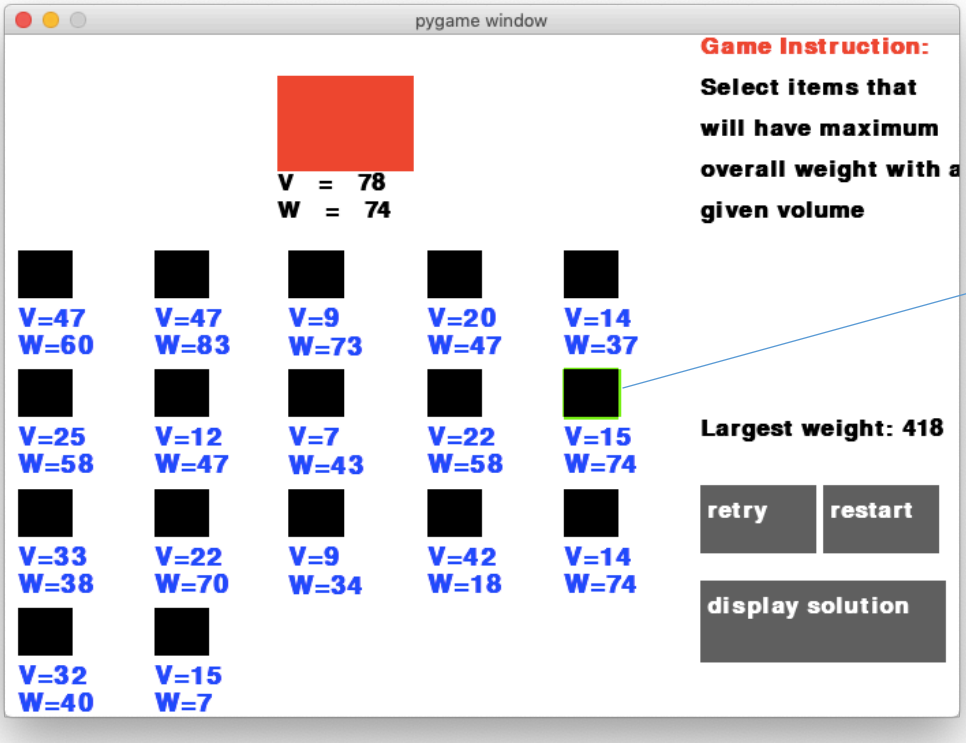
In the second puzzle, some extra buttons with functions specific to this puzzle are added, including the back button that enables the user to go back one step when he/she makes a mistake; the all edges button can give a peek of the layout of the graph with all edges displayed.



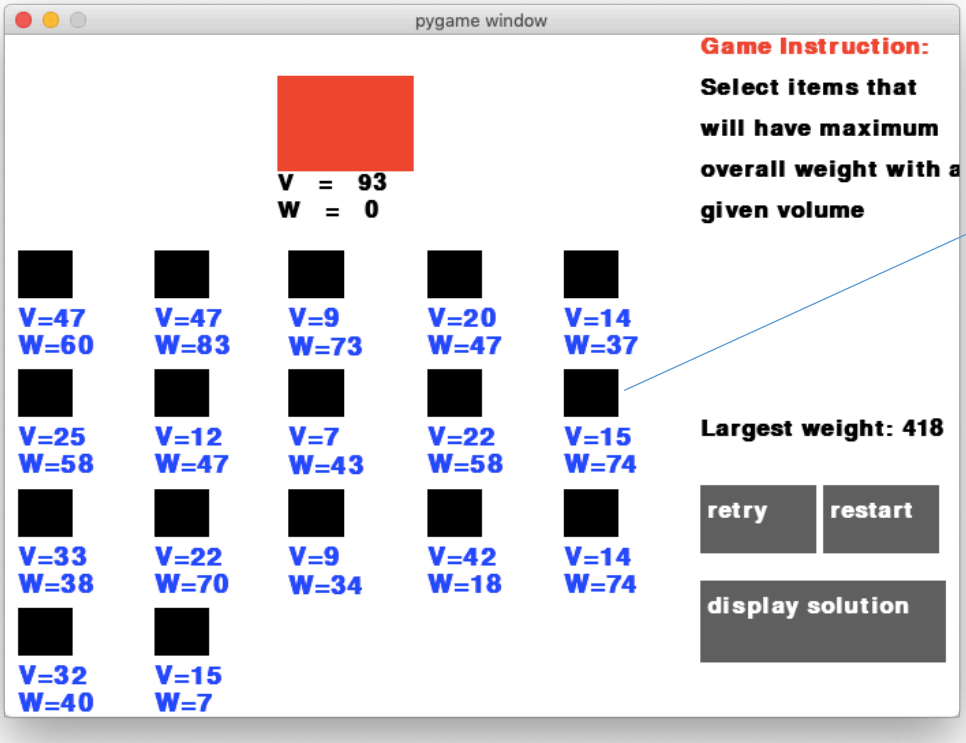
Effect of the all edge button: (may not be as useful when the number of edges become large)



In the third puzzle, there are not any extra buttons, but the ability of right clicking to deselect items also makes it easier for the user to undo moves.



An item has been selected.



It will then be deselected if one right clicks it.

In terms of how this game can be improved to make it more usable in the future, one of the “could” requirements is worth considering: extra guidance provided if a user does not understand the problem. Although game instructions are displayed on the righthand side of each puzzle, it is still possible that the user does not understand the description, in which case some extra explanations are required, maybe a few examples with illustrations. This can be achieved using some pop-up windows and question-mark-like buttons so that once it is clicked, a more detailed explanation of the current game can be displayed and help the user gain more information of the game.

Limitations and maintenance

Issues

Two major limitations of this game, which were mentioned at the analysis stage, are the lack of formal theories and fancy visual effect.

The lack of theories is determined by a characteristic of this game, which is to provide an interactive game to learners who want to learn some of the advanced algorithms, and there are lots of detailed explanation of how these algorithms work mathematically and theoretically, and therefore this would not be included in this game, which means this game is not a comprehensive learning tool and needs to be used along with some theory books, etc.

Secondly, fancy visual effects are not included in this game. Some of the coding websites provide good visual effects that can give users the experience of playing real games. But again, this is not the main focus of this game and learners may feel that this game is not as interesting as some of those online learning platforms.

In terms of the maintenance of this program, as it has been tested that OO programming is used throughout, it is relatively easy to extend the game in the future by taking advantage of the inheritance and relationships that have already been established. However, there are also some maintenance issues with this game, if it were to be developed in the future.

Firstly, the code logic might experience change in the future. During my development cycle, there has been several times of change of code logic, e.g. the relationship between different classes, although it has been modified several times and is now much more consistent than before. It is still likely that better logic will be developed in the future and that can cause problems as this means many changes to current codes would need to be made.

Secondly, the enlargement of the screen can cause some problems. Currently, the size of the screen is decided arbitrarily and so are the positions of elements on the screen. Lots of numerical values are used as position parameters. When the screen is enlarged, those parameters will be adjusted again in order to fit in the new screen.

Lastly, as puzzles are relatively independent of each other, it is not easy to reuse codes if new puzzles were to be developed in the future, especially puzzles of new styles. For example, if a new path-finding algorithm is going to be introduced, then the Node class can be reused. However, if some algorithms that involves computational geometry are introduced, then not many current codes can be reused.

How to address them

There are some ways to deal with the limitations and issues mentioned above.

In order to make this game more comprehensive in terms of its educational content, some references can be included, with the permission of authors, to give users some indication of what to look at if they want to study formally the theory behind these games. This can make the game more helpful to some of the beginners in this area.

Better visual effects can always be included if needed. As mentioned in the future development, some pictures can be used to replace squares to make the game more interesting, and other effects as well such as smooth move of the player, etc.

In terms of maintenance, first of all, the use of numerical values as position parameters needs to be changed to variables. This makes the enlargement of the screen easier and even a resizable screen could be developed in the future. Moreover, it is also important to have a detailed plan of what kind of puzzles will be included, as the code logic will need to be considered carefully before adding new things so that better logic can be applied first. This could also potentially make the code more reusable if OOP was used properly in the code.

Appendix

Bibliography

https://www.codingame.com/start
http://cs-playground-react.surge.sh/
https://educators.brainpop.com/2014/09/26/6-free-games-teaching-computer-programming-kids/
https://uva.onlinejudge.org/
https://leetcode.com/
http://codeforces.com/
http://www.bebras.uk/students.html
https://www.daysofwonder.com/tickettoride/en/

Final code

edu_game.py

```
import pygame as pg
import random
import copy
import math
import queue

# tuples declared for colours
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)
GREEN = (0, 255, 0)
DARKGREY = (169, 169, 169)
GREY = (96, 96, 96)
RED = (255, 0, 0)
BLUE = (0, 0, 255)

INF = 2**31  # infinity

all_sprites_group = pg.sprite.Group()
pg.init()
font = pg.font.SysFont('Calibri', 25, True, False)
clock = pg.time.Clock()
screenSize = (700, 500)
screen = pg.display.set_mode(screenSize)


class Element(pg.sprite.Sprite):  # this class is the parent class of all
relevant classes in the game
    def __init__(self, pos, size, color):
        super().__init__()
        self.image = pg.Surface(size)
        self.image.fill(color)
        self.centre = copy.deepcopy(pos)
        self.pos = [pos[0]-size[0]/2,pos[1]-size[1]/2]
        # pos parameter will be the top-left corner of the shape
        self.ori_pos = copy.deepcopy(self.pos)
        self.rect = self.image.get_rect(topleft=self.pos)
        self.size = size

    def tracking_event(self, keys): # this method will be called in the main
```

```
program loop to track any inputs
        pass

    def move(self, end_pos):  # some elements may be able to move
        pass

    def set_pos(self, pos):
        self.rect.x = pos[0]-self.size[0]/2
        self.rect.y = pos[1]-self.size[1]/2

    def set_size(self, size):
        self.size = size

    def get_size(self):
        return self.size

    def get_pos(self):
        return self.pos

    def reset(self):
        pass


class Tile(pg.sprite.Sprite):   # tile is specific to level1
    def __init__(self, pos, size):
        super().__init__()
        self.image = pg.Surface(size)
        self.rect = self.image.get_rect(topleft=pos)
        self.color = 1
        self.centre = [pos[0]+size[0]/2,pos[1]+size[1]/2]

    def set_color(self, color):
        # this is just a representation, color=1 means that the tile does
not need to be filled,
        # color=0 means it should be filled
        self.color = color

    def get_color(self):
        return self.color

    def get_centre(self):
        return self.centre

    def get_rect(self):
```

```
        return self.rect


class Button(object):  # button class, blueprint for all buttons
    def __init__(self, pos, size, color, word):
        self.image = pg.Surface(size)
        self.rect = self.image.get_rect(topleft=pos)
        self.image.fill(color)
        self.word = word
        self.size = size

    def is_over(self):  # returns whether the mouse is over the button, not
necessarily a click
        mouse_pos = pg.mouse.get_pos()
        return self.rect.x < mouse_pos[0] < self.rect.x+self.size[0] and
self.rect.y < mouse_pos[1] < \
            self.rect.y+self.size[1]

    def is_pressed(self):  # returns if the button is pressed
        return self.is_over() and pg.mouse.get_pressed()[0]  # mouse needs
to be over a certain button

    def display(self):  # display word and the button on the screen
        screen.blit(self.image, self.rect)
        screen.blit(font.render(self.word, True, WHITE), [self.rect.x+5,
self.rect.y+10])

    def switch(self, color):  # switch the color of the button
        self.image.fill(color)

    def update(self):  # monitors if the button has been clicked
        if self.is_over():
            self.switch(DARKGREY)
        else:
            self.switch(GREY)
        self.display()


class Puzzle(object):  # super class of all puzzles, like a virtual class
    def __init__(self):
        self.solution = -1
        self.button_list = []
        self.solution_list = []
```

```
    def initialise(self):
        pass

    def get_solution(self):  # find the optimum solution of a problem
        pass

    def display_info(self):  # display necessary information of the game,
such as life, time steps
        pass

    def restart(self):  # start the game again
        pass

    def retry(self):    # reset the game while keeping the map the same
        pass

    def display_solution(self):  # display game information and instructions
        pass

    def button_function(self):  # make sure buttons are updated
        self.restart()
        self.retry()
        self.display_solution()
        for b in self.button_list:
            b.update()

    def pre_update(self):  # pre_update will run outside the main program
loop
        self.initialise()
        self.get_solution()

    def update(self):  # update is called in the main loop
        pass


class PuzzleSelection(Puzzle):  # the puzzle selection interface

    def __init__(self):
        super().__init__()
        self.my_player = Element([0, 0], [0, 0], BLACK)
        self.puzzle1 = Button([50, 50], [180, 50], GREY, "Save the
princess")
        self.button_list.append(self.puzzle1)
        self.puzzle2 = Button([50, 200], [180, 50], GREY, "Shortest path")
```

```
        self.button_list.append(self.puzzle2)
        self.puzzle3 = Button([50, 350], [180, 50], GREY, "Knapsack
problem")
        self.button_list.append(self.puzzle3)

    def display_info(self):  # buttons and letters are displayed
        font1 = pg.font.SysFont('Calibri', 35, True, False)
        screen.blit(font1.render("Welcome", True, BLUE),[280,10])
        screen.blit(font.render("More challenges", True, BLACK),[360,350])
        screen.blit(font.render("coming soon...",True,BLACK),[360,380])

    def button_function(self):  # update buttons, called in the update
method
        global curPuzzle
        if self.puzzle1.is_pressed():
            curPuzzle = Puzzle1()
            curPuzzle.pre_update()
        elif self.puzzle2.is_pressed():
            curPuzzle = Puzzle2()
            curPuzzle.pre_update()
        elif self.puzzle3.is_pressed():
            curPuzzle = Puzzle3()
            curPuzzle.pre_update()

        for b in self.button_list:
            b.update()

    def update(self):
        self.button_function()
        self.display_info()


class Puzzle1(Puzzle):  # the save the princess puzzle
    maze = []
    princess_cor = [0, 0]
    maze_num = 0
    tile_list = []    # tile_list[i][j] means the tile with cor (i,j)

    def __init__(self):
        super().__init__()
        self.retry_button = Button([screenSize[1] + 10, 400 + 10], [70, 40],
GREY, "retry")
        self.button_list.append(self.retry_button)
        self.restart_button = Button([600 + 10, 400 + 10], [70, 40], GREY,
```

```
"restart")
        self.button_list.append(self.restart_button)
        self.solution_button = Button([500 + 10, 450 + 10], [170, 30], GREY,
"display solution")
        self.button_list.append(self.solution_button)
        self.my_player = Player1([0, 0], [0, 0], BLACK, 0)

    def initialise(self):
        # 1=wall 2=player 3=princess
        all_sprites_group.empty()
        Puzzle1.maze_num = random.randrange(5, 30)  # number of maze are
randomly generated
        Puzzle1.maze = [[0] * Puzzle1.maze_num for i in
range(Puzzle1.maze_num)]
        Puzzle1.tile_list = [[Tile([0,0],[0,0])] * Puzzle1.maze_num for i in
range(Puzzle1.maze_num)]

        nSpecialElement = random.randrange(0,
                                           int(Puzzle1.maze_num *
Puzzle1.maze_num * 0.5))
        # randrange [a,b), 50% of the Puzzle1.maze is wall
        player_cor = random.randrange(0, Puzzle1.maze_num *
Puzzle1.maze_num)

        #  assign different numbers to the player, princess and tiles
        Puzzle1.maze[player_cor // Puzzle1.maze_num][player_cor %
Puzzle1.maze_num] = 2
        while True:  # generate the coordinate of the target
            princess_cor = random.randrange(0, Puzzle1.maze_num *
Puzzle1.maze_num)
            if princess_cor != player_cor:
                break

        Puzzle1.maze[princess_cor // Puzzle1.maze_num][princess_cor %
Puzzle1.maze_num] = 3
        nSpecialElement -= 2

        # generate walls
        for i in range(nSpecialElement):
            while True:
                wall_cor = random.randrange(0, Puzzle1.maze_num *
Puzzle1.maze_num)
                if wall_cor != princess_cor and wall_cor != player_cor:
                    break
```

```
            Puzzle1.maze[wall_cor // Puzzle1.maze_num][wall_cor %
Puzzle1.maze_num] = 1

        side_length = 500 / Puzzle1.maze_num
        # iterate through the maze and generate relevant objects
        for i in range(Puzzle1.maze_num):
            for j in range(Puzzle1.maze_num):
                t = Tile((j * side_length, i * side_length), (side_length,
side_length))

                if Puzzle1.maze[i][j] == 1:
                    t.set_color(0)

                elif Puzzle1.maze[i][j] == 2:
                    size = [side_length * 0.4,
                            side_length * 0.4]  # the size of the player
will make up two fifths of a tile
                    self.my_player = Player1(t.get_centre(), size, BLUE, [i,
j])  # player is centred
                    all_sprites_group.add(self.my_player)

                elif Puzzle1.maze[i][j] == 3:
                    size = [side_length * 0.4, side_length * 0.4]
                    princess = NPC(t.get_centre(), size, RED, [i,j])
                    all_sprites_group.add(princess)
                    Puzzle1.princess_cor = [i, j]

                Puzzle1.tile_list[i][j] = t

    def get_solution(self):
        # BFS to find the solution
        self.solution = -1
        self.solution_list = []
        q = []
        q_head=0
        q_tail=0
        # every element in q is a list of three integers s[0]: row num,
s[1]:column number;
        # s[2]: number of steps, s[3]: father
        # father is used to backtrack to find the path
        visited = [[0] * Puzzle1.maze_num for i in range(Puzzle1.maze_num)]
        # array used to tell if a grid has been visited
        dir = [[0, 1, 0, -1], [1, 0, -1, 0]]
        start_pos = self.my_player.get_cor()
```

```
        end_pos = Puzzle1.princess_cor
        q.append([start_pos[0], start_pos[1], 0, -1])
        q_tail += 1
        # bfs implementation
        while q_head!=q_tail:
            s = q[q_head]
            if [s[0], s[1]] == end_pos:
                self.solution=s[2]
                father = q_head
                while True:
                    self.solution_list.append([q[father][0],q[father][1]])
                    father=q[father][3]
                    if father==-1:
                        break
                self.solution_list.reverse()
                break
            for i in range(4):
                new_r = s[0] + dir[0][i]
                new_c = s[1] + dir[1][i]
                if new_r < 0 or new_r >= Puzzle1.maze_num or new_c < 0 or
new_c >= Puzzle1.maze_num or Puzzle1.maze[new_r][
                    new_c] == 1 or visited[new_r][new_c]:
                    continue
                q.append([new_r, new_c, s[2] + 1,q_head])
                q_tail+=1
                visited[new_r][new_c] = 1
            q_head += 1

    @staticmethod
    def draw_tiles():  # draw all tiles onto the screen in the tile_list
        for i in range(Puzzle1.maze_num):
            for j in range(Puzzle1.maze_num):
                t = Puzzle1.tile_list[i][j]
                pg.draw.rect(screen,BLACK,t.get_rect(),t.get_color()*1)

    def display_info(self):
        # game instruction
        game_instruction = []
        game_instruction.append(font.render("Game Instruction:", True, RED))
        game_instruction.append(font.render("Move the player to", True,
BLACK))
        game_instruction.append(font.render("meet the princess", True,
BLACK))
        game_instruction.append(font.render("in minimum number", True,
```

```
BLACK))
        game_instruction.append(font.render("of moves", True, BLACK))
        for i in range(len(game_instruction)):
            screen.blit(game_instruction[i],[500+10,i*30])

        # game information
        if self.my_player.get_cor()==Puzzle1.princess_cor and
self.my_player.get_step()==self.solution:
            screen.blit(font.render("You Win!", True, RED), [500 + 10, 200 +
10])

screen.blit(font.render("Congratulations",True,RED),[500+10,200+10+30])
        elif self.my_player.get_cor()==Puzzle1.princess_cor:
            screen.blit(font.render("Well done!", True, RED), [500 + 10, 200
+ 10])
            screen.blit(font.render("Try to do it with", True, RED), [500 +
10, 200 + 10 + 30])
            screen.blit(font.render("fewer moves", True, RED), [500 + 10,
200 + 10 + 30*2])

        screen.blit(font.render("steps taken: " +
str(self.my_player.get_step()), True, BLACK), [500 + 10, 300 + 10])
        screen.blit(font.render("steps required: " + str(self.solution),
True, BLACK), [500 + 10, 300 + 10+30])

    def retry(self):    # reset the player position and the maze stays
unchanged
        if self.retry_button.is_pressed():
            self.my_player.reset()

    def restart(self):  # the maze is re-generated
        if self.restart_button.is_pressed():
            all_sprites_group.empty()
            self.tile_list=[]
            self.pre_update()

    def display_solution(self):
        if self.solution_button.is_pressed():
            for i in range(len(self.solution_list) - 1):
                s1 = self.solution_list[i]
                s2 = self.solution_list[i + 1]
                t1 = Puzzle1.tile_list[s1[0]][s1[1]]
                t2 = Puzzle1.tile_list[s2[0]][s2[1]]
                pg.draw.line(screen, GREEN, t1.get_centre(),
```

```
t2.get_centre(), 3)

    def update(self):
        all_sprites_group.draw(screen)
        Puzzle1.draw_tiles()
        self.button_function()
        self.display_info()


class NPC(Element):  # mainly the princess class
    def __init__(self,pos, size, color, cor):
        Element.__init__(self, pos, size, color)
        self.cor = cor
        self.ori_cor = copy.deepcopy(cor)  # any ori_cor must use deepcopy


class Player1(Element):  # class Player1 is a friend of class Puzzle1

    def __init__(self, pos, size, color, cor):
        Element.__init__(self, pos, size, color)
        self.cor = cor
        self.ori_cor = copy.deepcopy(cor)  # any ori_cor must use deepcopy
        self.step = 0

    def tracking_event(self,keys):
        if keys == pg.K_RIGHT:
            self.move([self.cor[0], self.cor[1]+1])
        elif keys == pg.K_LEFT:
            self.move([self.cor[0], self.cor[1] -1])
        elif keys == pg.K_UP:
            self.move([self.cor[0]-1, self.cor[1]])
        elif keys == pg.K_DOWN:
            self.move([self.cor[0]+1, self.cor[1]])

    def move(self, end_cor):
        new_r=end_cor[0]
        new_c=end_cor[1]
        flag = (0 <= new_r < Puzzle1.maze_num) and (0 <= new_c <
Puzzle1.maze_num) and (Puzzle1.maze[new_r][new_c] != 1)
        # check if the player reaches the boundary and if the block is
reachable
        if flag:
            # find the new centre of the player
            newCentre = Puzzle1.tile_list[new_r][new_c].get_centre()
```

```
            newX = newCentre[0] - self.size[0] / 2
            newY = newCentre[1] - self.size[1] / 2
            # change the attributes of the player
            self.rect.x = newX
            self.rect.y = newY
            self.cor[0] = new_r
            self.cor[1] = new_c
            self.step += 1

    def set_cor(self, cor):
        self.cor = cor

    def get_cor(self):
        return self.cor

    def get_step(self):
        return self.step

    def reset(self):
        self.cor=copy.deepcopy(self.ori_cor) # any ori_cor must be deeply
copied
        self.rect.x=self.ori_pos[0]
        self.rect.y=self.ori_pos[1]
        self.step = 0


class Node(pg.sprite.Sprite):   # node is specific to level2
    def __init__(self,pos,size,num,color):
        super().__init__()
        self.radius = size
        self.centre = pos
        self.color = color
        self.num = num
        self.weight = 0

    def is_mouse_over(self):
        mouse_pos = pg.mouse.get_pos()
        return self.centre[0]-self.radius < mouse_pos[0] < self.centre[0] +
self.radius and \
            self.centre[1]-self.radius < mouse_pos[1] < self.centre[1] +
self.radius

    def get_centre(self):
        return self.centre
```

```
    def get_size(self):
        return self.radius

    def get_color(self):
        return self.color

    def set_weight(self,w):
        self.weight=w

    def get_num(self):
        return self.num

    def get_weight(self):
        return self.weight

    def __lt__(self,other):  # operator< overload
        return self.weight < other.weight


class Player2(Element):
    def __init__(self,pos,size,color, cor):
        Element.__init__(self, pos, size, color)
        self.cor = cor
        self.ori_cor = copy.deepcopy(cor)  # any ori_cor must use deepcopy
        self.path = 0

    def tracking_event(self,button):
        pass
        if button == 1:  # left key pressed
            for n in Puzzle2.node_list:
                if n.is_mouse_over():
                    self.move(n)

    def move(self,node):
        flag = False
        for n in Puzzle2.graph[self.cor]:  # find if node is connected
            if n.num == node.num:
                flag = True
                break

        if flag:
            Puzzle2.visited_node.append(node)
            self.path+=node.weight
```

```
            self.cor = node.num
            p=node.get_centre()
            newX = p[0]-self.size[0]/2
            newY = p[1]-self.size[1]/2
            self.rect.x = newX
            self.rect.y = newY

    def move_back(self):  # called when the back button is pressed
        if len(Puzzle2.visited_node) > 1:
            n2 = Puzzle2.visited_node.pop()
            n1 = Puzzle2.visited_node[-1]
            self.cor = n1.num
            p = n1.get_centre()
            newX = p[0] - self.size[0] / 2
            newY = p[1] - self.size[1] / 2
            self.rect.x = newX
            self.rect.y = newY
            for n in Puzzle2.graph[n1.num]:
                if n2.num == n.num:
                    self.path -= n.weight
                    break

    def set_cor(self, cor):
        self.cor = cor

    def get_cor(self):
        return self.cor

    def reset(self):
        self.cor = copy.deepcopy(self.ori_cor)  # any ori_cor must be deeply
copied
        self.rect.x = self.ori_pos[0]
        self.rect.y = self.ori_pos[1]
        self.path=0

    def get_path(self):
        return self.path


class Puzzle2(Puzzle):  # the shortest path puzzle
    node_num = 0
    graph = []    # graph is adjacency list where each element is
[nodeNum,weight]
    node_list = []
```

```
    visited_node = []
    princess_cor = 0

    def __init__(self):
        super().__init__()
        self.my_player=Player2([0,0] ,[0,0],BLUE, 0)
        self.retry_button=Button([screenSize[1] + 10, 350], [85, 40], GREY,
"retry")
        self.button_list.append(self.retry_button)
        self.restart_button=Button([screenSize[1] + 100, 350], [85, 40],
GREY, "restart")
        self.button_list.append(self.restart_button)
        self.back_button=Button([screenSize[1] + 10, 400], [85, 40], GREY,
"back")
        self.button_list.append(self.back_button)
        self.all_edges_button=Button([screenSize[1] + 100, 400], [88, 40],
GREY, "all edges")
        self.button_list.append(self.all_edges_button)
        self.solution_button = Button([screenSize[1] + 10, 450], [180, 40],
GREY, "display solution")
        self.button_list.append(self.solution_button)

    def initialise(self):
        Puzzle2.node_num = random.randrange(3, 30)
        # generate the position of every node
        num = int(math.sqrt(Puzzle2.node_num)) + 1
        sep = 500/num  # separation of each node, 500 is the sceen size
        i = 0
        j = 0
        min_r = 500
        for k in range(Puzzle2.node_num):
            start_x = int((j+0.1)*sep)  # leave some blank space
            end_x = int((j+0.9)*sep)
            start_y = int((i+0.1)*sep)
            end_y = int((i+0.9)*sep)
            circle_pos = [random.randint(int(start_x+sep*0.2),int(end_x-
sep*0.2)),random.randint(int(start_y+sep*0.2),int(end_y-sep*0.2))]
            # make sure the node is not too small
            radius = min(circle_pos[0]-start_x,end_x-
circle_pos[0],circle_pos[1]-start_y,end_y-circle_pos[1])
            min_r = min(radius,min_r)
            Puzzle2.node_list.append(Node(circle_pos,radius,k,DARKGREY))
            j += 1
            if j == num:
```

```
                i += 1
                j %= num

        # generate the graph in an adjacency list
        Puzzle2.graph = [[] for i in range(Puzzle2.node_num)]
        for i in range(Puzzle2.node_num):
            used = [0] * (Puzzle2.node_num + 1)  # edges that have already
been connected
            used[i] = 1
            out_degree = random.randrange(1, int(Puzzle2.node_num * 0.8))
            # number of edges connected to the current node
            j = 0
            while j < out_degree:
                n = random.randrange(0, Puzzle2.node_num)
                if used[n]:
                    continue
                used[n] = 1
                j += 1
                w = random.randrange(1, 100)  # random weight of the edge
                node = Puzzle2.node_list[n]
                node.set_weight(w)
                Puzzle2.graph[i].append(node)

        # initialise the player
        n1 = Puzzle2.node_list[random.randrange(0, Puzzle2.node_num)]
        self.my_player=Player2(n1.get_centre(),[min_r,min_r],BLUE, n1.num)
        Puzzle2.visited_node.append(n1)
        all_sprites_group.add(self.my_player)

        # generate the position of the target that is not the same as the
player
        while True:
            n2 = Puzzle2.node_list[random.randrange(0, Puzzle2.node_num)]
            if n2.num != n1.num:

princess=NPC(n2.get_centre(),[n2.get_size(),n2.get_size()],RED, n2.num)
                all_sprites_group.add(princess)
                Puzzle2.princess_cor=n2.num
                break

    @staticmethod
    def draw_nodes():
        for i in range(Puzzle2.node_num):
            n = Puzzle2.node_list[i]
```

```
            pg.draw.circle(screen,n.get_color(),n.get_centre(),n.get_size())

    @staticmethod
    def draw_visited_edges():
        for i in range(len(Puzzle2.visited_node)-1):
            n1 = Puzzle2.visited_node[i]
            n2 = Puzzle2.visited_node[i+1]
            pg.draw.line(screen,GREEN,n1.get_centre(),n2.get_centre(),3)

    def draw_edges(self):  # draw the edge if the player is on the node or
if the mouse is over
        for n in Puzzle2.node_list:
            if n.is_mouse_over():
                self.draw_edges_from_node(n)

        n=Puzzle2.node_list[self.my_player.get_cor()]
        self.draw_edges_from_node(n)

    @staticmethod
    def draw_all_edges():  # called when all edges button pressed
        for i in range(Puzzle2.node_num):
            for j in range(len(Puzzle2.graph[i])):
                n1=Puzzle2.node_list[i]
                n2=Puzzle2.graph[i][j]
                pg.draw.aaline(screen,GREEN,n1.get_centre(),n2.get_centre())
                # p1 = n1.get_centre()
                # p2 = n2.get_centre()
                # screen.blit(font.render(str(n2.weight), True, BLACK),
[(p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2])

    @staticmethod
    def draw_edges_from_node(node):  # draw the edge from a node
        font = pg.font.SysFont('Calibri', 20, True, False)
        for i in range(len(Puzzle2.graph[node.num])):
            n1=node
            n2=Puzzle2.graph[n1.num][i]
            p1 = n1.get_centre()
            p2 = n2.get_centre()

screen.blit(font.render(str(n2.weight),True,BLACK),[(p1[0]+p2[0])/2,(p1[1]+p
2[1])/2])
            # the weight is at the middle point of a edge
            pg.draw.aaline(screen, GREEN, p1,p2)
```

```
    def get_solution(self):  # Dijkstra's to find the optimum solution of a
problem
        self.solution_list = []
        visited = [0]*Puzzle2.node_num
        dist = [INF]*Puzzle2.node_num
        prev = [0]*Puzzle2.node_num  # backtrack list of nodes
        pq = queue.PriorityQueue()
        nd = copy.deepcopy(Puzzle2.node_list[self.my_player.get_cor()])
        nd.set_weight(0)
        pq.put(nd)
        prev[nd.num] = -1
        dist[nd.num] = 0
        while not pq.empty():
            nd = pq.get()
            if visited[nd.num]:
                continue
            if nd.num == Puzzle2.princess_cor:
                self.solution = dist[nd.num]
                father = nd.num
                while True:
                    self.solution_list.append(father)
                    if prev[father] == -1:
                        break
                    father = prev[father]
                self.solution_list.reverse()
                break
            visited[nd.num] = 1
            for n in Puzzle2.graph[nd.num]:
                if visited[n.num]:
                    continue
                if dist[n.num] > dist[nd.num] + n.weight:
                    dist[n.num] = dist[nd.num] + n.weight
                    prev[n.num] = nd.num
                    pq.put(n)

    def display_info(self):  # display necessary information of the game,
such as life, time steps
        # game instruction
        game_instruction = []
        game_instruction.append(font.render("Game Instruction:", True, RED))
        game_instruction.append(font.render("Click a node to move", True,
BLACK))
        game_instruction.append(font.render("the player to meet", True,
BLACK))
```

```
        game_instruction.append(font.render("the princess by the", True,
BLACK))
        game_instruction.append(font.render("shortest path", True, BLACK))
        for i in range(len(game_instruction)):
            screen.blit(game_instruction[i], [500 + 10, i * 30])

        # game information
        if self.my_player.get_cor() == Puzzle2.princess_cor and
self.my_player.get_path() == self.solution:
            screen.blit(font.render("You Win!", True, RED), [screenSize[1] +
10, 200 + 10])
            screen.blit(font.render("Congratulations", True, RED),
[screenSize[1] + 10, 200 + 10 + 30])
        elif self.my_player.get_cor() == Puzzle2.princess_cor:
            screen.blit(font.render("Well done!", True, RED), [screenSize[1]
+ 10, 200 + 10])
            screen.blit(font.render("Try to do it with", True, RED),
[screenSize[1] + 10, 200 + 10 + 20])
            screen.blit(font.render("fewer moves", True, RED),
[screenSize[1] + 10, 200 + 10 + 20 * 2])

        screen.blit(font.render("path length: " +
str(self.my_player.get_path()), True, BLACK), [screenSize[1] + 10, 275 +
10])
        screen.blit(font.render("shortest path: " + str(self.solution),
True, BLACK), [screenSize[1] + 10, 275 + 10 + 30])

    def restart(self):  # start the game again
        if self.restart_button.is_pressed():
            all_sprites_group.empty()
            Puzzle2.graph = []
            Puzzle2.node_list = []
            Puzzle2.visited_node = []
            self.pre_update()
            self.my_player.reset()

    def back(self):  # function of the back button
        if self.back_button.is_pressed():
            self.my_player.move_back()

    def retry(self):  # reset the game while keeping the map the same
        if self.retry_button.is_pressed():
            Puzzle2.visited_node = []
            self.my_player.reset()
```

```
    def display_all_edges(self):
        if self.all_edges_button.is_pressed():
            Puzzle2.draw_all_edges()

    def display_solution(self):
        if self.solution_button.is_pressed():
            for i in range(len(self.solution_list)-1):
                n1=Puzzle2.node_list[self.solution_list[i]]
                n2=Puzzle2.node_list[self.solution_list[i+1]]
                pg.draw.line(screen, RED, n1.get_centre(), n2.get_centre(),
3)

    def update(self):
        self.draw_nodes()
        self.draw_edges()
        self.draw_visited_edges()
        self.display_info()
        self.button_function()
        self.display_all_edges()
        self.back()
        all_sprites_group.draw(screen)


class Item(Element):  # items in the third puzzle
    def __init__(self,pos,size,color,v,w):
        Element.__init__(self, pos, size, color)
        self.volume = v
        self.weight = w
        self.selected = False

    def get_weight(self):
        return self.weight

    def get_volume(self):
        return self.volume

    def is_mouse_over(self):
        mousePos = pg.mouse.get_pos()
        return self.centre[0]-self.size[0]/2 < mousePos[0] < self.centre[0]
+ self.size[0]/2 and \
            self.centre[1]-self.size[1]/2 < mousePos[1] < self.centre[1] +
self.size[1]/2
```

```
    def mouse_pressed(self): # detects if the item is left or right clicked
        if self.is_mouse_over():
            if pg.mouse.get_pressed()[0]:
                return 1
            elif pg.mouse.get_pressed()[2]:
                return 2
        return 0

    def clicked(self):
        if self.mouse_pressed() == 1 and not self.selected:  # left button
            self.selected = True
            return 1
        elif self.mouse_pressed() == 2 and self.selected:  # right button
            self.selected = False
            return -1
        else:
            return 0  # not clicked

    def highlight(self,color):
        pg.draw.rect(screen, color, [self.pos, self.size], 4)

    def deselect(self):
        self.selected = False


class Bag(Element):  # the bag in the third puzzle
    def __init__(self, pos, size, color, v):
        Element.__init__(self, pos, size, color)
        self.volume=v
        self.ori_v = v
        self.weight=0

    def set_weight(self,w):
        self.weight=w

    def set_volume(self,v):
        self.volume=v

    def get_volume(self):
        return self.volume

    def get_weight(self):
        return self.weight
```

```
    def reset(self):
        self.weight = 0
        self.volume = self.ori_v


class Puzzle3(Puzzle):  # knapsack problem
    item_num = 0
    item_list = []

    def __init__(self):
        super().__init__()
        self.my_player=Bag([250,65],[100,70],RED,random.randint(10,100))
        all_sprites_group.empty()
        self.retry_button = Button([screenSize[1] + 10, 330], [85, 50],
GREY, "retry")
        self.button_list.append(self.retry_button)
        self.restart_button = Button([screenSize[1] + 100, 330], [85, 50],
GREY, "restart")
        self.button_list.append(self.restart_button)
        self.solution_button = Button([screenSize[1] + 10, 400], [180, 60],
GREY, "display solution")
        self.button_list.append(self.solution_button)

    def initialise(self):
        Puzzle3.item_num=random.randrange(4, 30)
        self.my_player = Bag([250, 65], [100, 70], RED, random.randint(10,
100))
        all_sprites_group.add(self.my_player)

        # calculate the number of items per row and column
        num_x = int(math.sqrt(10 / 7 * Puzzle3.item_num)) + 1
        num_y = int(math.sqrt(7 / 10 * Puzzle3.item_num)) + 1
        sep_x = 500/num_x
        sep_y = 350/num_y

        i = 0
        j = 0

        # give each item its position, size, color, volume, weight, etc
        for k in range(Puzzle3.item_num):
            start_x = int((j+0.3)*sep_x)
            end_x = int((j+0.7)*sep_x)
            start_y = 150+int((i + 0.3) * sep_y)
            end_y = 150 + int((i + 0.7) * sep_y)
```

```
            item_pos = [start_x,start_y]
            size = [end_x-start_x,end_y-start_y]
            v = random.randrange(5,50)
            w = random.randrange(1,100)
            item = Item(item_pos,size,BLACK,v,w)
            Puzzle3.item_list.append(item)
            all_sprites_group.add(item)
            j += 1
            if j == num_x:
                i += 1
                j %= num_x

    def display_weight_and_volume(self):
        # the bag
        p = self.my_player.get_pos()
        s = self.my_player.get_size()
        pos = [p[0], p[1]+s[1]]
        font = pg.font.SysFont('Calibri', 25, True, False)
        screen.blit(font.render("V   =   "+str(self.my_player.get_volume()),
True, BLACK), pos)
        screen.blit(font.render("W   =
"+str(self.my_player.get_weight()),True,BLACK), [pos[0],pos[1]+20])

        # items
        for item in Puzzle3.item_list:
            p = item.get_pos()
            s = item.get_size()
            pos = [p[0], p[1]+s[1]+5]
            font = pg.font.SysFont('Calibri', int(s[0]*0.7), True, False)
            screen.blit(font.render("V=" + str(item.get_volume()), True,
BLUE), pos)
            screen.blit(font.render("W=" + str(item.get_weight()), True,
BLUE), [pos[0], pos[1] + s[0]*0.5])

    def item_highlight(self):  # show items that are selected
        for item in Puzzle3.item_list:
            added = item.clicked()
            if item.selected:
                item.highlight(GREEN)

            # update the value of the bag
            if added == 1:
                self.my_player.weight += item.get_weight()
                self.my_player.volume -= item.get_volume()
```

```
            elif added == -1:
                self.my_player.weight -= item.get_weight()
                self.my_player.volume += item.get_volume()

    def get_solution(self):  # dynamic porgramming used to optimum solution
of a problem
        w = [0]*(Puzzle3.item_num + 1)
        v = [0]*(Puzzle3.item_num + 1)

        # get the weight and volume of all items and store them in the
arrays
        for i in range(1, Puzzle3.item_num + 1):
            w[i] = Puzzle3.item_list[i-1].get_weight() # i-1 is because
index starts from 0 in the item_list
            v[i] = Puzzle3.item_list[i-1].get_volume()

        dp = [[0] * (self.my_player.volume+1) for i in
range(Puzzle3.item_num + 1)]
        # array used to store dp results
        choices = [[[]] * (self.my_player.volume + 1) for i in
range(Puzzle3.item_num + 1)]
        # store how items are selected for dp[i][j]
        for i in range(1, Puzzle3.item_num + 1):
            for j in range(1,self.my_player.get_volume()+1):
                dp[i][j] = dp[i-1][j]
                choices[i][j] = choices[i-1][j]
                if j >= v[i] and dp[i-1][j-v[i]]+w[i] > dp[i-1][j]:
                    dp[i][j] = dp[i-1][j-v[i]]+w[i]
                    choices[i][j] = choices[i-1][j-v[i]] + [i]

        self.solution = dp[Puzzle3.item_num][self.my_player.get_volume()]
        self. solution_list =
choices[Puzzle3.item_num][self.my_player.get_volume()]

    def display_info(self):  # display necessary information of the game,
such as life, time steps
        # game instruction
        game_instruction = []
        game_instruction.append(font.render("Game Instruction:", True, RED))
        game_instruction.append(font.render("Select items that ", True,
BLACK))
        game_instruction.append(font.render("will have maximum", True,
BLACK))
        game_instruction.append(font.render("overall weight with a", True,
```

```
BLACK))
        game_instruction.append(font.render("given volume", True, BLACK))
        for i in range(len(game_instruction)):
            screen.blit(game_instruction[i], [500 + 10, i * 30])

        # game information
        if self.my_player.get_weight() == self.solution:
            screen.blit(font.render("You Win!", True, RED), [500 + 10, 200 +
10])
            screen.blit(font.render("Congratulations", True, RED), [500 +
10, 200 + 10 + 30])

        if self.my_player.get_volume() < 0:
            screen.blit(font.render("Whoops! Right lick ", True, RED), [500
+ 10, 200 + 10])
            screen.blit(font.render("selected item again", True, RED), [500
+ 10, 200 + 10 + 20])
            screen.blit(font.render("to deselect it", True, RED), [500 + 10,
200 + 10 + 20 + 20])

        screen.blit(font.render("Largest weight: " + str(self.solution),
True, BLACK), [500 + 10, 280])

    def restart(self):  # start the game again
        if self.restart_button.is_pressed():
            all_sprites_group.empty()
            Puzzle3.item_list = []
            self.pre_update()
            self.my_player.reset()

    def retry(self):  # reset the game while keeping the map the same
        if self.retry_button.is_pressed():
            for item in Puzzle3.item_list:
                item.deselect()
            self.my_player.reset()

    def display_solution(self):  # called when the solution is pressed
        if self.solution_button.is_pressed():
            for s in self.solution_list:
                Puzzle3.item_list[s-1].highlight(RED)

    def update(self):
        self.display_weight_and_volume()
        self.display_info()
```

```
        self.item_highlight()
        self.button_function()
        all_sprites_group.draw(screen)


done = False
curPuzzle = PuzzleSelection()
curPuzzle.pre_update()

while not done:  # the main program loop
    for event in pg.event.get():
        if event.type == pg.QUIT:
            done=True
        elif event.type == pg.KEYDOWN:  # if the key is pressed
            curPuzzle.my_player.tracking_event(event.key)
        elif event.type == pg.MOUSEBUTTONDOWN:  # if the mouse is pressed
            curPuzzle.my_player.tracking_event(event.button)

    screen.fill(WHITE)
    curPuzzle.update()   # this line must be after the group draw code

    pg.display.flip()
    clock.tick(60)
```