# Analysis

## Problem identification

There already exists lots of educational games of different kinds that can give beginners in computer science a brief introduction to what is computer science, 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 sorts 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 lies 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 due to 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 to the learner.

## Stakeholders

Despite every effort is 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 of 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 stakeholders:

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 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.

## 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 who 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 chock-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 Kids Computer Programming

There are lots of introductory games for kids 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 cars that enable them to claim railway routes connecting cities in various countries around the world.

Students through the missions that they choose about connecting one city to another, 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 committing to the visualisation of algorithmic problems, though with various approaches and levels. For instance, coding game has fascinating game context for new coders and makes it much more appealing to beginners, while online judging systems provides much more challenging and well-designed problems and feedback on users’ solutions after they submit their codes. 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 thought, 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 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 simplified 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 problems 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 of my solution. The first one is that the formal theories behind each game is not included in this game and users will have to find out themselves how exactly 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 systematic learning. 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 of further study on 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 concentration of this game. Some of the games may have already done the abstraction and may be presented to the user in a different way from how its original description is. This could potentially make the game less interesting to play for some of the users.

## Requirements

|  |  |  |  |
| --- | --- | --- | --- |
| No. | Description | Comments | priority |
| Design | | | |
|  | Clear instruction on how to play the game | So that a user can easily start to crack on the game | Must |
|  | Player movement | The player is able to move | Must |
|  | All information required is displayed | So that the user is able to play the game | Must |
|  | Game running correctly and logically | e.g. illegal moves are not allowed | Must |
|  | Problem illustration as straightforward as possible | Many of games in my project will be a simplified version of some rather complicated problems, so it is important to give clear illustration | Must |
|  | 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 |
|  | 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 extra help available | Could |
|  | Challenges are offered in various levels of difficulty | So that users can build up knowledge and understanding of an algorithm | Should |
|  | Some illustrations of characters are included | To add more fun to the learning process | Could |
|  |  |  |  |
|  |  |  |  |
| User experience | | | |
|  | Game can run smoothly and provides good user experience | Smooth running can give the user the willingness to keep playing | Should |
|  | Program does not contain serious bugs | So that the game will not crush unexpectedly in most situations | Should |
|  | 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 | | | |
|  | 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 |
|  | The game is as interactive as possible | So that user will not give up the game easily if he/she is stuck | Could |
|  | 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 |
|  | Solutions are straightforward for players to follow | Although explanations on how a solution is achieved will not be included, the correct answer will be displayed and made easy for a user to follow | Should |
|  | 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 |
|  | Stories are interesting to most players | Again, this can make a user enjoy more about the learning process | Could |
|  | 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 | | | |
|  | Idea of OO 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 |
|  | Code is as elegant as possible | So that is would be easier for others and to read | Could |
|  | Code is nicely commented | To make review process easier and more efficient | Could |
|  |  |  |  |
| Stakeholders’ requirement | | |  |
|  | Puzzles are carefully selected so that they are neither too obscure nor too basic. | By basic, they refer to algorithms such as binary search, while the network flow would be considered too advanced | Must |
|  | Puzzles are designed to deepen the understanding of algorithms rather than broaden the knowledge |  | Must |
|  | Puzzles are designed to as interesting as possible with some illustrations available | For example, some pictures could be used | Could |
| Others | | | |
|  | 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. |  |
|  | 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 |  |
|  | Python interpreter | This is used to run the source code of this game. |  |
|  | 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: breath-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

Game project

Puzzle 3

Puzzle 2

Puzzle 1

Puzzle 1

Maze, player and target generated

Maze can be of different sizes

Player blocked by wall and boundry

Movement of the player

Game information displayed

Button functions

Solution displayed graphically

Bfs to calculate the solution

Puzzle 2

Graph, player and target generated

Display edge when the mouse is over or the player is on

Edges and nodes drawn

Game information displayed

Button functions

Solution displayed graphically

Dijkstra’s algrithm to calculate the solution

Maze can be of different sizes

Node weight shown at the centre of the edge

Path taken highlighted

Puzzle 3

Bag and items generated

Random generation of volume and weight

Right click to deselect items

Left click to select items

Game information displayed

Button functions

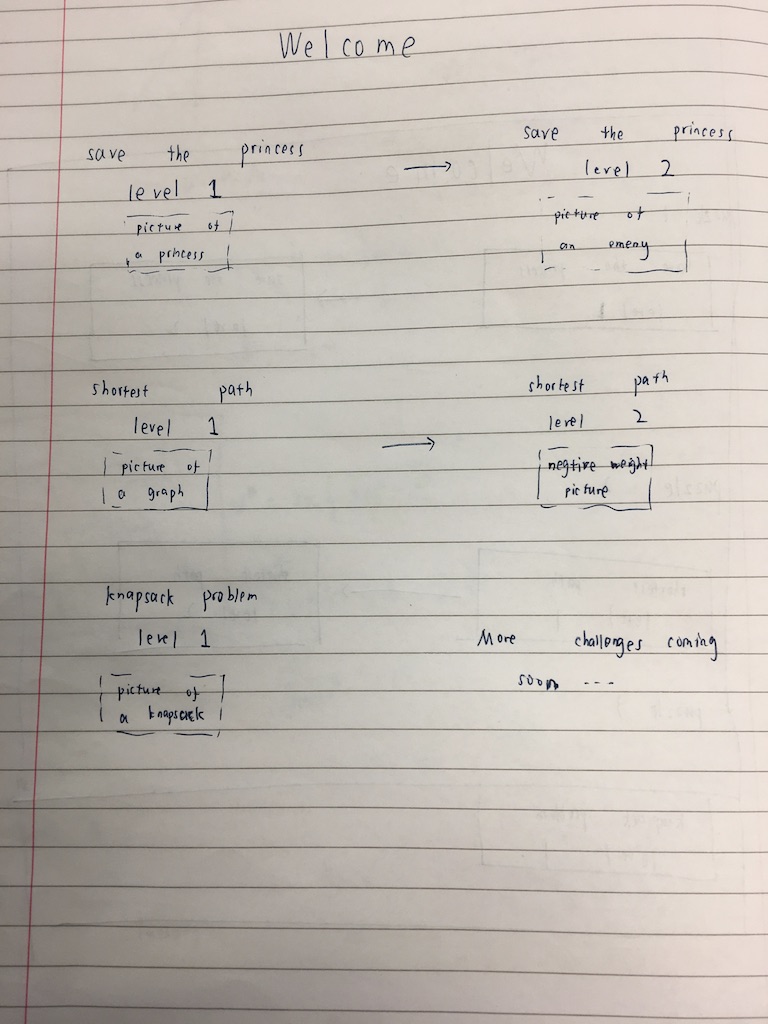
Solution displayed graphically

DP used to calculate the solution

Highlight selected items

## Interface design

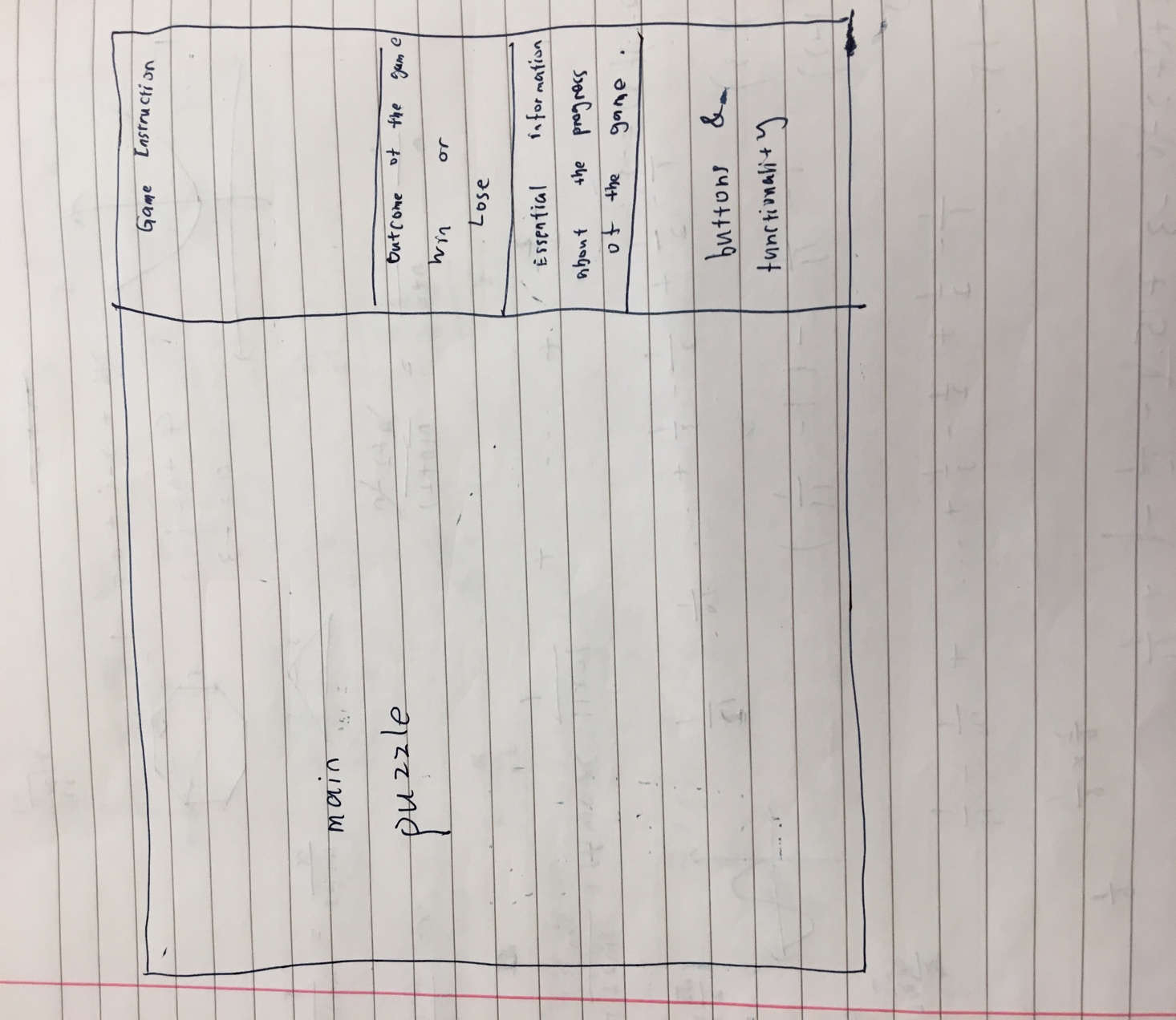
### Level Selection



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 that particular level. This is included such 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 level



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 of the game which includes four parts:

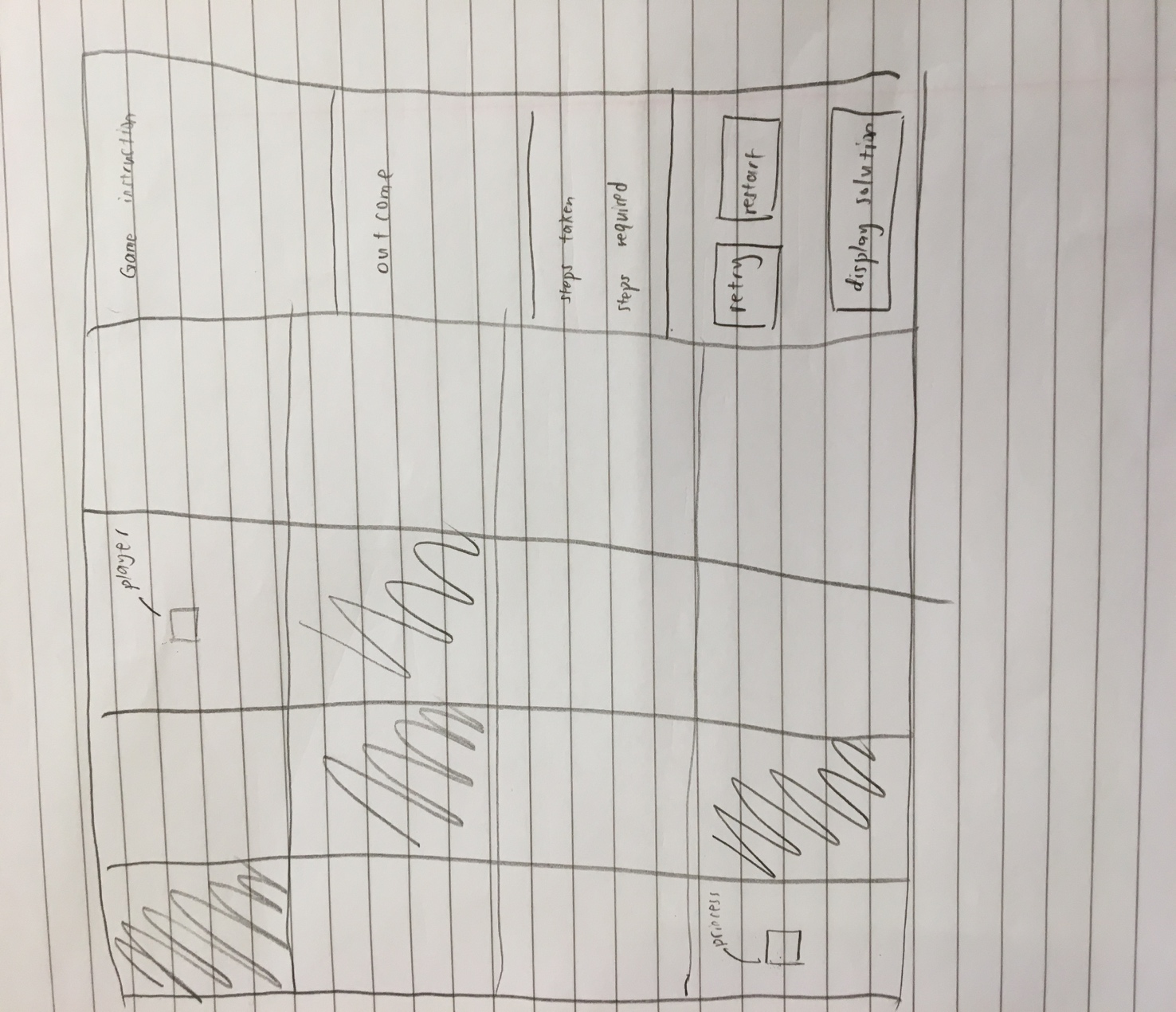
* Game instruction: a brief introduction to the user on the objective of the game and how one could play the game;
* Outcome of the game: it shows whether the user has successfully accomplished the objective or there is a better solution to the game;
* Current progress of the game and the optimum solution: 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;

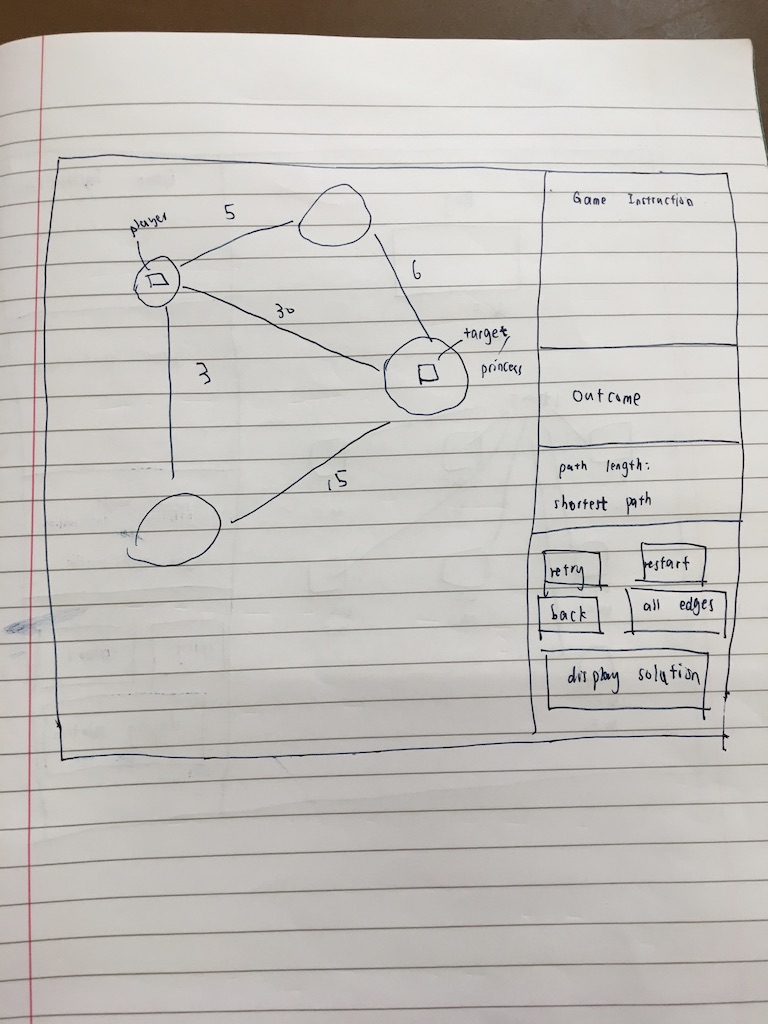
Display solution – solution of the puzzle will be displayed if this button is pressed.

### Save the princess



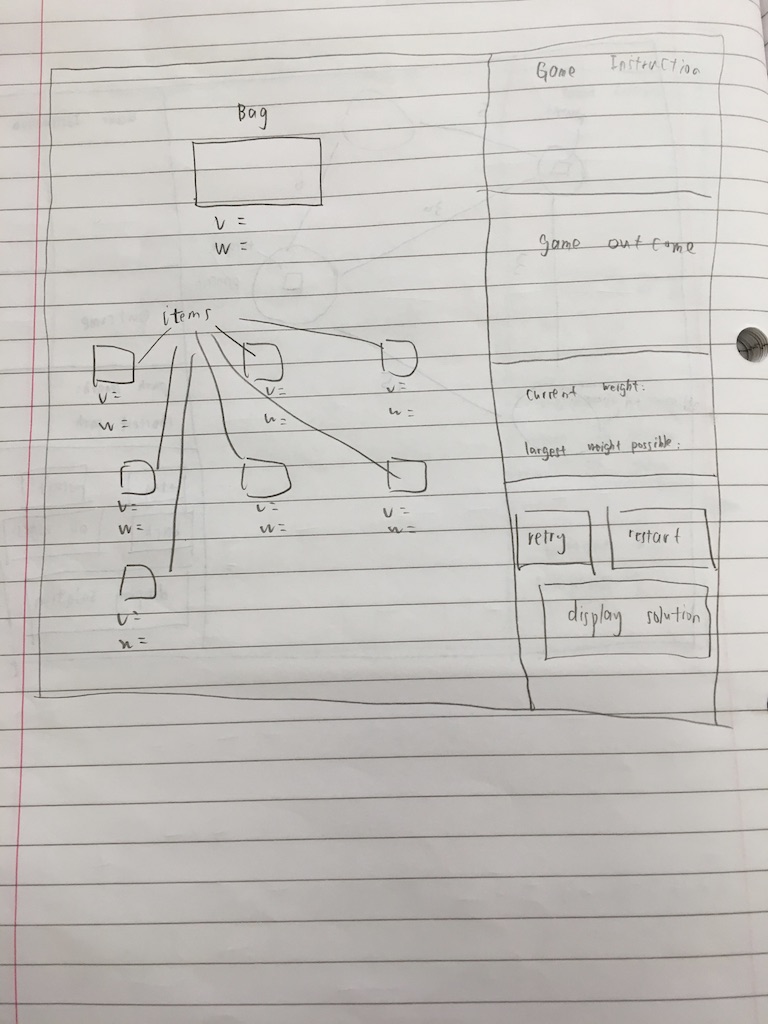
This is the “save the princess” challenge level 1, which is a classic application of breath-first search algorithm. The user needs to move the player to reach the princess in minimum number of steps. The maze consists of a number of cells where shaded ones are not reachable. Therefore, the player has to move across the maze and pass only those white squares. On the right-hand side, basic information such as how many steps the user has taken and what is the minimum number of steps one could 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.

### 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. This 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 is the optimum solution.

### Knapsack problem



Again, the knapsack problem is 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.

## 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 instruction 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 on 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 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 to not only 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 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 a general level and the relationship between different puzzles, the following inheritance diagram is created accordingly:

Puzzle

Puzzle1

Puzzle2

Puzzle3

### Class diagram

Element

Pygame.sprite.Sprite

Bag

Item

Player2

Player1

NPC

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

Player1

-step

+get\_step()

Player2

-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()

Level

-solution

-buttion\_list

+initialise()

+get\_solution()

+display\_info()

+restart()

+retry()

+pre\_update()

+update()

Level1

+maze

+princess\_cor

+maze\_num

+tile\_list

-retry\_button

-restart\_button

-solution\_button

-my-player

-solution\_list

+draw\_tiles()

Level2

+node\_num

+graph

+node\_list

+visited\_node

+princess\_cor

-my\_player

-retry\_button

-restart\_button

-back\_button

-all\_edges\_button

-solution\_button

+draw\_nodes()

+draw\_visited\_node()

+draw\_edges()

+draw\_all\_edges()

+draw\_edges\_from\_node()

+back()

+display\_all\_edges()

Level3

+item\_num

+item\_list

-my\_player

-retry\_button

-restart\_button

-solution\_button

+display\_weight\_and\_volume()

+item\_highlight()

Button

-image

-rect

-word

-size

+is\_over()

+is\_pressed()

+display()

+switch(color)

+update()

Node

-radius

-centre

-color

-num

-weight

+is\_mouse\_over()

+get\_centre()

+get\_size()

+get\_color()

+get\_weight()

+get\_num

+set\_weight(w)

Tile

-image

-rect

-color

-centre

+set\_color(color)

+get\_color

+get\_centre()

+get\_rect()

## 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 level classes and player classes both inherit from a general “level” and a “player” class respectively, only algorithms specific to a level will be demonstrated in the pseudocode.

All level classes consist of two major functions: game elements generation and optimum solution finding and they are designed as follows:

### Puzzle 1: save the princess

The initialise method allows the maze of uzzle 1 as well as the player and the princess to be generated ready for to be displayed. It is also possible to use this method to re-generate these elements for a new challenge.

1. procedure initialise()
2. # 1=wall 2=player 3=princess
3. maze\_num = random(5, 30)
4. maze = [maze\_num, maze, nun]
5. tile\_list = []
7. nSpecialElement = random(0, maze\_num ^ 2 \* 0.5)
8. # randrange [a,b), 60mod of the maze is wall
9. playerCor = random(0, maze\_num ^ 2)
10. maze[playerCor // maze\_num][playerCor mod maze\_num] = 2
12. while **True**
13. princessCor = random(0, maze\_num \* maze\_num)
14. if princessCor != playerCor
15. break
16. endif
17. endwhile
18. maze[princessCor // maze\_num][princessCor mod maze\_num] = 3
19. nSpecialElement -= 2
21. for i= 0 to nSpecialElement
23. while **True**
24. wallCor = random(0, maze\_num \* maze\_num)
25. if wallCor != princessCor and wallCor != playerCor
26. break
27. endif
28. endwhile
29. maze[wallCor // maze\_num][wallCor mod maze\_num] = 1
30. next i
32. sideLength = 500 / maze\_num
33. for i = 0 to maze\_num
34. for j = 0 to maze\_num
36. t = new Tile((j \* sideLength, i \* sideLength), (sideLength, sideLength))
38. if maze[i][j] == 1
39. t.set\_color(0)
41. else if maze[i][j] == 2
42. size = [sideLength \* 0.4,
43. sideLength \* 0.4]  # the size of the player will make up two fifths of a tile
44. my\_player = new Player1(t.get\_centre(), [i, j], size, BLUE)  # player is centred
45. all\_sprites\_group.add(my\_player)
47. else if maze[i][j] == 3
48. size = [sideLength \* 0.4, sideLength \* 0.4]
49. princess = new NPC(t.get\_centre(), [i, j], size, RED)
50. all\_sprites\_group.add(princess)
51. princess\_cor = [i, j]
53. tile\_list[i][j] = t
54. endif
55. next i
56. procedure

The optimum solution is pre-calculated after the maze is generated and before the game starts. This is calculated in the get\_solution method. In this specific challenge, breath-first search is used to find the optimum solution.

### Puzzle 2: shortest path

1. procedure get\_solution()
2. solution = -1
3. solution\_list = []
4. q = []
5. qHead = 0
6. qTail = 0
7. # every element in q is a list of three integers s[0] row num, s[1]column number;
8. # s[2] number of steps, s[3] father
9. visited = [maze\_num,maze\_num]
10. dir = [[0, 1, 0, -1], [1, 0, -1, 0]]
11. startPos = my\_player.get\_cor()
12. endPos = princess\_cor
13. q.append([startPos[0], startPos[1], 0, -1])
14. qTail += 1
15. while qHead != qTail
16. s = q[qHead]
17. if [s[0], s[1]] == endPos
18. solution = s[2]
19. father = qHead
20. while **True**
21. solution\_list.append([q[father][0], q[father][1]])
22. father = q[father][3]
23. if father == -1
24. break
25. solution\_list.reverse()
26. break
27. endif
28. for i = 0 to 3
29. new\_r = s[0] + dir[0][i]
30. new\_c = s[1] + dir[1][i]
31. if new\_r < 0 or new\_r >= maze\_num or new\_c < 0 or new\_c >= maze\_num or maze[new\_r][
32. new\_c] == 1 or visited[new\_r][new\_c]
33. continue
34. endif
35. q.append([new\_r, new\_c, s[2] + 1, qHead])
36. qTail += 1
37. visited[new\_r][new\_c] = 1
38. qHead += 1
39. next i
40. endwhile
41. endprocedure

The puzzle2 initialise has a similar function to that of the puzzle1: generating the graph and the player.

1. procedure initialise()
2. node\_num = random(3, 30)
3. # generate the position of every node
4. num = sqrt(node\_num) + 1
5. sep = 500 / num
6. i = 0
7. j = 0
8. minR = 500
9. for k =0 to node\_num
10. start\_x = int((j + 0.1) \* sep)  # leave some blank space
11. end\_x = int((j + 0.9) \* sep)
12. start\_y = int((i + 0.1) \* sep)
13. end\_y = int((i + 0.9) \* sep)
14. circlePos = [random(int(start\_x + sep \* 0.2), int(end\_x - sep \* 0.2)),
15. random(int(start\_y + sep \* 0.2), int(end\_y - sep \* 0.2))]
16. # make sure the node is not too small
17. radius = min(circlePos[0] - start\_x, end\_x - circlePos[0], circlePos[1] - start\_y, end\_y - circlePos[1])
18. minR = min(radius, minR)
19. node\_list.append(new Node(circlePos, radius, k, DARKGREY))
20. j += 1
21. if j == num
22. i += 1
23. j %= num
24. endif
25. next k
27. # generate the graph in a adjacency list
28. graph = [node\_num]
29. for i =0 to node\_num
30. used = [node\_num]
31. used[i] = 1
32. outDegree = random(1, int(node\_num \* 0.8))
33. j = 0
34. while j < outDegree
35. n = random(0, node\_num)
36. if used[n]
37. continue
38. endif
39. used[n] = 1
40. j += 1
41. w = random(1, 100)
42. node = node\_list[n]
43. node.set\_weight(w)
44. graph[i].append(node)
45. endwhile
46. next i
48. # initialise the player
49. n1 = node\_list[random(0, node\_num)]
50. my\_player =new Player2(n1.get\_centre(), n1.num, [minR, minR], BLUE)
51. visited\_node.append(n1)
52. all\_sprites\_group.add(my\_player)
53. while **True**
54. n2 = node\_list[random(0, node\_num)]
55. if n2.num != n1.num
56. princess =new NPC(n2.get\_centre(), n2.num, [n2.get\_size(), n2.get\_size()], RED)
57. all\_sprites\_group.add(princess)
58. princess\_cor = n2.num
59. break
60. endif
61. endwhile
63. end procedure

The optimum solution in this level is calculated through the get\_solution method and the Dijkstra’s algorithm is used.

1. procedure get\_solution()  # find the optimum solution of a problem
2. solution\_list = []
3. visited = [0] \* node\_num
4. dist = [INF] \* node\_num
5. prev = [0] \* node\_num
6. pq = PriorityQueue()
7. nd = node\_list[my\_player.get\_cor()]
8. nd.set\_weight(0)
9. pq.put(nd)
10. prev[nd.num] = -1
11. dist[nd.num] = 0
12. while not pq.empty()
13. nd = pq.pop()
14. if visited[nd.num]
15. continue
16. endif
17. if nd.num == princess\_cor
18. solution = dist[nd.num]
19. father = nd.num
20. while **True**
21. solution\_list.append(father)
22. if prev[father] == -1
23. break
24. endif
25. father = prev[father]
26. endwhile
27. solution\_list.reverse()
28. break
29. endif
30. visited[nd.num] = 1
31. for i = 0 to len(graph[nd.num])
32. n = graph[nd.num][i]
33. if visited[n.num]
34. continue
35. endif
36. if dist[n.num] > dist[nd.num] + n.weight
37. dist[n.num] = dist[nd.num] + n.weight
38. prev[n.num] = nd.num
39. pq.enQueue(n)
40. endif
41. next i
42. endwhile
43. end procedure

### Puzzle 3: knapsack problem

Similarly, the initialise method is used to generate the bag and items in this puzzle:

1. procedure initialise()
2. item\_num = random(4, 30)
3. my\_player = new Bag([250, 65], [100, 70], RED, random.randint(10, 100))
4. all\_sprites\_group.add(my\_player)
5. num\_x = int(sqrt(10 / 7 \* item\_num)) + 1
6. num\_y = int(sqrt(7 / 10 \* item\_num)) + 1
7. sep\_x = 500 / num\_x
8. sep\_y = 350 / num\_y
9. i = 0
10. j = 0
11. for k = 0 to item\_num-1
12. start\_x = int((j + 0.3) \* sep\_x)
13. end\_x = int((j + 0.7) \* sep\_x)
14. start\_y = 150 + int((i + 0.3) \* sep\_y)
15. end\_y = 150 + int((i + 0.7) \* sep\_y)
16. item\_pos = [start\_x, start\_y]
17. size = [end\_x - start\_x, end\_y - start\_y]
18. v = random(5, 50)
19. w = random(1, 100)
20. item = new Item(item\_pos, size, BLACK, v, w)
21. item\_list.append(item)
22. all\_sprites\_group.add(item)
23. j += 1
24. if j == num\_x
25. i += 1
26. j = j mod num\_x
27. endif
28. next k
29. endprocedure

Dynamic programming is the technique used here to calculate the solution, details as follows:

1. procedure get\_solution()  # find the optimum solution of a problem
2. w = [0] \* (item\_num + 1)
3. v = [0] \* (item\_num + 1)
4. for i = 1 to item\_num
5. w[i] = item\_list[i - 1].get\_weight()  # i-1 is because index starts from 0 in the item\_list
6. v[i] = item\_list[i - 1].get\_volume()
7. next i
8. dp = [my\_player.volume + 1,item\_num + 1]
9. choices = [my\_player.volume + 1, item\_num + 1]
10. # store how items are selected for dp[i][j]
11. for i =1 to item\_num
12. for j =1 to my\_player.get\_volume()
13. dp[i][j] = dp[i - 1][j]
14. choices[i][j] = choices[i - 1][j]
15. if j >= v[i] and dp[i - 1][j - v[i]] + w[i] > dp[i - 1][j]
16. dp[i][j] = dp[i - 1][j - v[i]] + w[i]
17. choices[i][j] = choices[i - 1][j - v[i]] + [i]
18. endif
19. next j
20. next i
21. solution = dp[item\_num][my\_player.get\_volume()]
22. solution\_list = choices[item\_num][my\_player.get\_volume()]
24. endprocedure

## 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 | Design | Expected output |
|  | Level1 | | |
|  | Player movement | Arrow keys to move the player | Player moves accordingly |
|  | Clear instruction on how to play the game | Wining text displayed | Text are displayed when the game starts |
|  | Clear instruction on how to play the game | Steps taken change as player moves | As required by the design |
|  | 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 |
|  | Essential functions are implemented and available to the user | Restart button working | The game starts again with a different setting, |
|  | Essential functions are implemented and available to the user | Display solution button working | Solution displayed |
|  | Algorithm test | | |
|  | N/A | initialise method generates the maze and the player | The maze, player and target are ready when the game starts |
|  | N/A | 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 | | | |
|  | All information required is displayed | Edges and weights displayed | Text are displayed when the game starts |
|  | Player movement | Player able to move | Player moves when a node is clicked |
|  | Essential functions are implemented and available to the user | Edges displayed when mouse is over | As required in the design |
|  | Essential functions are implemented and available to the user | Path taken displayed | Trajectory is shown when the player moves |
|  | 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 |
|  | Clear instruction on how to play the game | Game information displayed | Text are displayed when the game starts |
|  | 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 |
|  | Essential functions are implemented and available to the user | Restart button working | The game starts again with a different setting, |
|  | Essential functions are implemented and available to the user | Display solution button working | Solution displayed |
|  | Essential functions are implemented and available to the user | Back button working | To undo the movement, click the undo button |
|  | Essential functions are implemented and available to the user | All edges button working | All edges are displayed when the button is clicked |
|  | Algorithm test | | |
|  | N/A | initialise method generates the maze and the player | The maze, player and target are ready when the game starts |
|  | N/A | 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 | | | |
|  | All information required is displayed | Bags and items displayed | Text are displayed when the game starts |
|  | All information required is displayed | Weight and volume underneath | Text are displayed when the game starts |
|  | Clear instruction on how to play the game | Game instruction shown | Text are displayed when the game starts |
|  | Clear instruction on how to play the game | Game information changes dynamically | correct information is displayed when the game progresses |
|  | 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 |
|  | Essential functions are implemented and available to the user | Restart button working | The game starts again with a different setting, |
|  | Essential functions are implemented and available to the user | Display solution button working | Solution displayed |
|  | Essential functions are implemented and available to the user | To select items | Left click to select items and selected items are highlighted in green |
|  | Essential functions are implemented and available to the user | To deselect items | Right click to deselect items and green lines will disappear |
|  | Algorithm test | | |
|  | N/A | initialise method generates the bag and items | The bag and items are ready when the game starts |
|  | N/A | 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

## 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 of these solving these problems are not particularly important if there is no dependency between them.

### Maze generation

The first step to be implemented is to generate the maze, player and the target. 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]

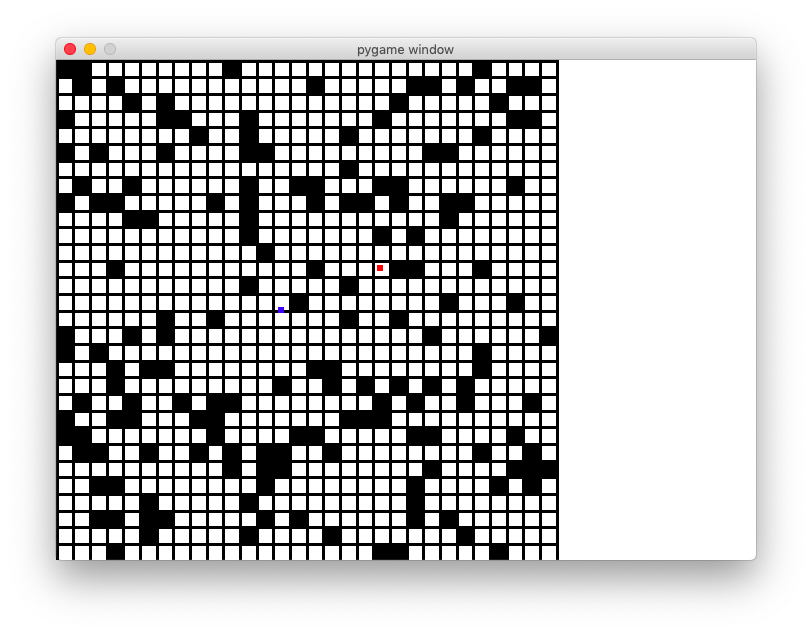
The program is then run to be tested, it is working properly when the size of the maze is not particularly large.

### Player movement

Secondly, the ability of the player of making movements is implemented as the method of the Player1 class, considering that this is one of the behaviours that the player1 should have. 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 is also included in the code the player will be prevented from 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

Again, the program is tested. However, this time, when the player is moving a maze with a large size, a problem arises: The player is no longer staying 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.

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

### Information display

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

**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])  
  
 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])

### 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()

### Solution in BFS

Finally, the correct solution of the puzzle will be calculated once the maze has been generated. Not only the number but also the graphical solution will be displayed to the user via the solution button. This is to reflect the requirement No.16 that the solution straightforward for the user to follow.

**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

### 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

## 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. This has also been included in the initialise method of class Puzzle2, which, like Puzzle1, inherits from the parent class Puzzle. The initialise method also overrides the method in the class Puzzle, so that polymorphism can be used to make the code more flexible.

**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**

### Edges and nodes display

The unique part of this puzzle2 is that involves 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. 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())

Edges are, by definition, connections between different nodes. So, one would naturally draw a line between the centres of two nodes. However, it is not easy to decide how the weights of each edge can 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.

Another feature that is also included to improve usability is that edges and their weights of a particular node will also be displayed when the player is in the node or the user places the mouse over the node. This is due to that not all the edges are displayed at first and the user might want to find out the weights of edges connected to a specific node. With this feature, they can do so by moving the mouse over a node.

**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)

**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]+p2[1])/2])  
 *# the weight is at the middle point of a edge* pg.draw.aaline(screen, GREEN, p1,p2)

### Information display

Similar to puzzle1, some information of the game needs to be displayed to help the player understand the game. 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])

### Button functions