Pathfinding Visualizer Design Report

Dung Tran – 103486496

Object-Oriented Programming

Swinburne University of Technology

I. Introduction

The program is designed as a custom program of Object-Oriented Programming course in semester 2. The program aims at delivering tool to visualize some famous Pathfinding Algorithms like Depth-First Search, Breadth-First Search and A Star Search. The design of the program is following Object-Oriented principles as well as some Design Patterns implementations

II. User Interface and Functionalities

A. *User Interface*

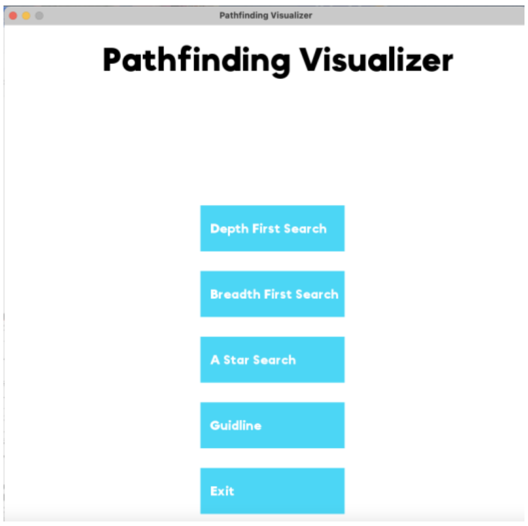
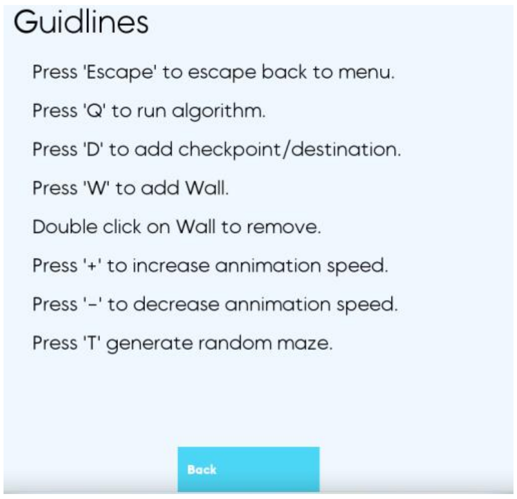


Figure 1 Menu Scene

The program has been written in C# which was using the SplashKitSDK as the graphical library of choice. Users are able to go to different scene and choosing different kind of algorithms.

Additionally, users could see the guidelines of the program by clicking on it. Therefore, it is easy to see which button to choose in terms of running the algorithm.



Guidlines

Press 'Escape' to escape back to menu.

Press 'Q' to run algorithm.

Press 'D' to add checkpoint/destination.

Press 'W' to add Wall.

Double click on Wall to remove.

Press '+' to increase annimation speed.

Press '-' to decrease annimation speed.

Press 'T' generate random maze.

Figure 2 Guidelines

The visualization interface has been displayed as a grid of 20x20. This is where users will use different button to customize the grid as well as running the algorithm.

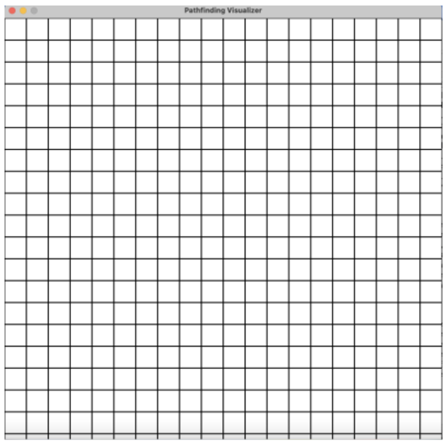


Figure 3 Grid

B. *Functionalities*

Users could customize the grid by adding wall, destination node or by setting random maze.

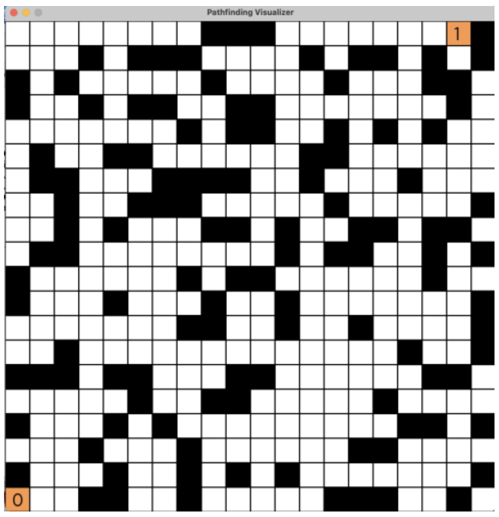


Figure 4 Random Maze Sample

By customizing the grid, the pathfinding visualizer can now come up with different shape of path. The process of finding the path could be slow down or boost the speed up based on “+” and “-“ button

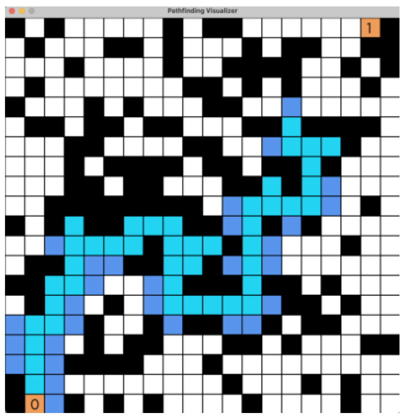


Figure 5 Finding Process

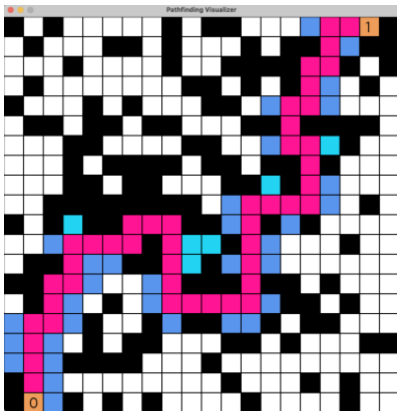


Figure 6 Successfully finding the path

III. Design of Program

A. *UML Design*

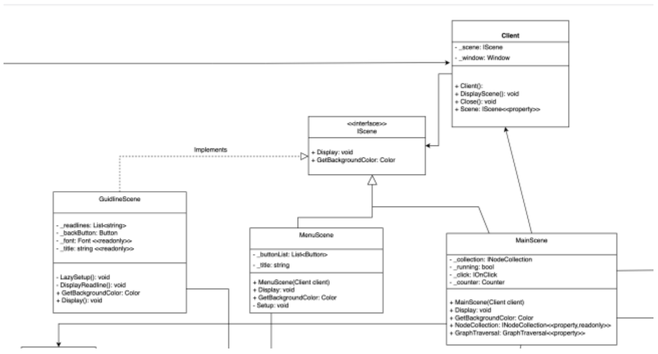

Client
- _scene: IScene
- _window: Window
+ Client():
+ DisplayScene(): void
+ Close(): void
+ Scene: IScene<<property>>
<<interface>>
IScene
+ Display: void
+ GetBackgroundColor: Color
Implements
GuidlineScene
- _readlines: List<string>
- _backButton: Button
- _font: Font <<readonly>>
- _title: string <<readonly>>
- LazySetup(): void
- DisplayReadline(): void
+ GetBackgroundColor: Color
+ Display(): void
MenuScene
- _buttonList: List<Button>
- _title: string
+ MenuScene(Client client)
+ Display: void
+ GetBackgroundColor: Color
- Setup: void
MainScene
- _collection: INodeCollection
- _running: bool
- _click: IOnClick
- _counter: Counter
+ MainScene(Client client)
+ Display: void
+ GetBackgroundColor: Color
+ NodeCollection: INodeCollection<<property,readonly>>
+ GraphTraversal: GraphTraversal<<property>>

Figure 7 Client Design

The Client controls every scene. Each scene will implement the abstraction of interface scene which is responsible for display and get color for the background of that scene. There are three main scenes which has been mentioned in the design. The full UML diagram could be viewed on “https://app.diagrams.net/#G1YuzgL-fsWnJPlMcT6if9TSRXIRvILk7Y”

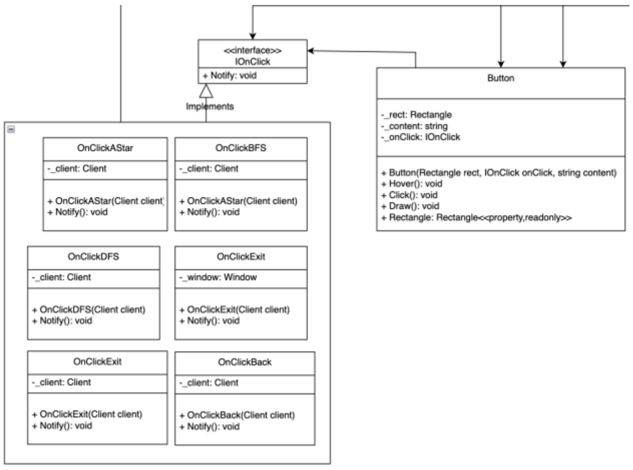


Figure 8 Button

The button is design as having shown in the user interface that it will control the event of clicking something related to the client. The button will rely on different kind of on click strategies to have distinct action of click.

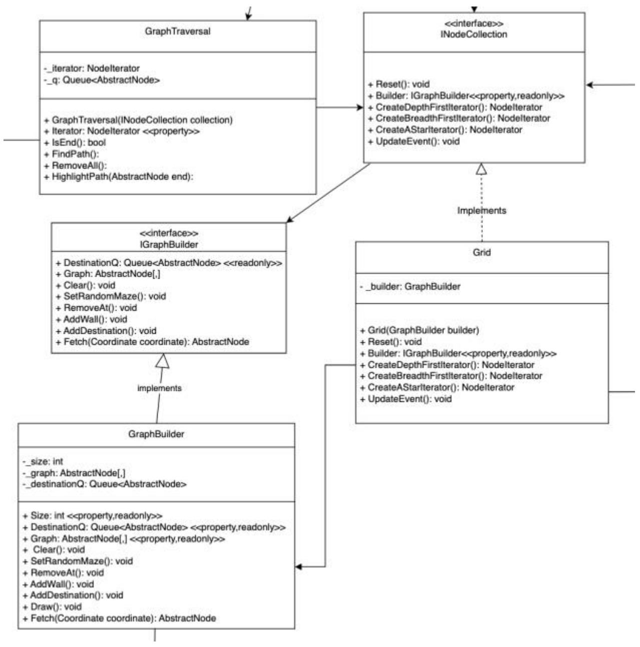

GraphTraversal
-_iterator: NodeIterator
-_q: Queue<AbstractNode>
+ GraphTraversal(INodeCollection collection)
+ Iterator: NodeIterator <<property>>
+ IsEnd(): bool
+ FindPath():
+ RemoveAll():
+ HighlightPath(AbstractNode end):
<<interface>>
INodeCollection
+ Reset(): void
+ Builder: IGraphBuilder<<property,readonly>>
+ CreateDepthFirstIterator(): NodeIterator
+ CreateBreadthFirstIterator(): NodeIterator
+ CreateAStarIterator(): NodeIterator
+ UpdateEvent(): void
Implements
<<interface>>
IGraphBuilder
+ DestinationQ: Queue<AbstractNode> <<readonly>>
+ Graph: AbstractNode[,]
+ Clear(): void
+ SetRandomMaze(): void
+ RemoveAt(): void
+ AddWall(): void
+ AddDestination(): void
+ Fetch(Coordinate coordinate): AbstractNode
Grid
- _builder: GraphBuilder
+ Grid(GraphBuilder builder)
+ Reset(): void
+ Builder: IGraphBuilder<<property,readonly>>
+ CreateDepthFirstIterator(): NodeIterator
+ CreateBreadthFirstIterator(): NodeIterator
+ CreateAStarIterator(): NodeIterator
+ UpdateEvent(): void
implements
GraphBuilder
-_size: int
-_graph: AbstractNode[,]
-_destinationQ: Queue<AbstractNode>
+ Size: int <<property,readonly>>
+ DestinationQ: Queue<AbstractNode> <<property,readonly>>
+ Graph: AbstractNode[,] <<property,readonly>>
+ Clear(): void
+ SetRandomMaze(): void
+ RemoveAt(): void
+ AddWall(): void
+ AddDestination(): void
+ Draw(): void
+ Fetch(Coordinate coordinate): AbstractNode

Figure 9 Graph Traversal class and Node Collection

The GraphBuilder is used to build and customize the graph based on what user want. For example, it is responsible for adding Destination or Wall to the graph while Grid implementing INodeCollection will be responsible for managing the builder. The Graph Traversal is used to process the pathfinding algorithm.

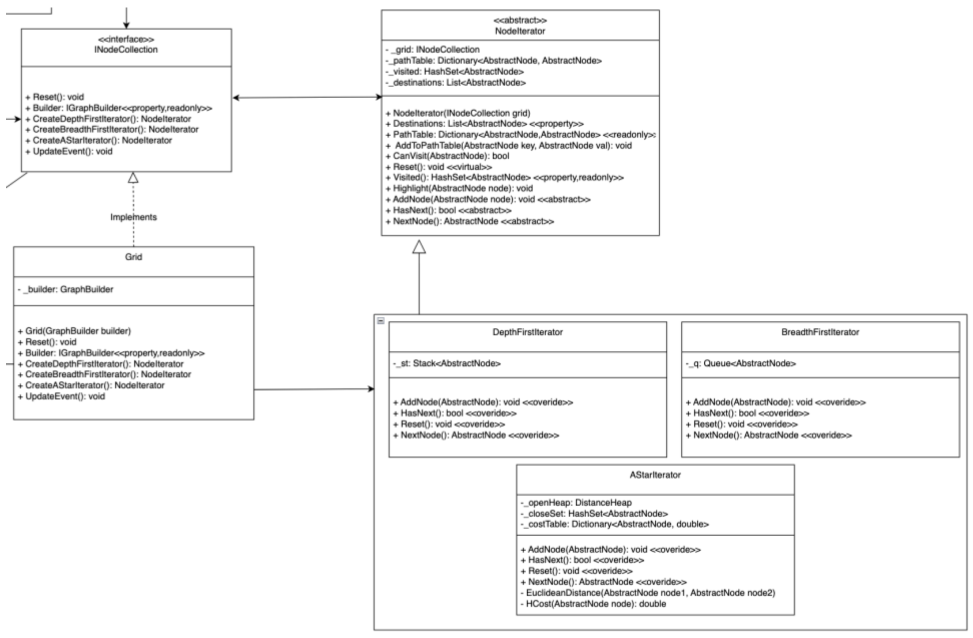
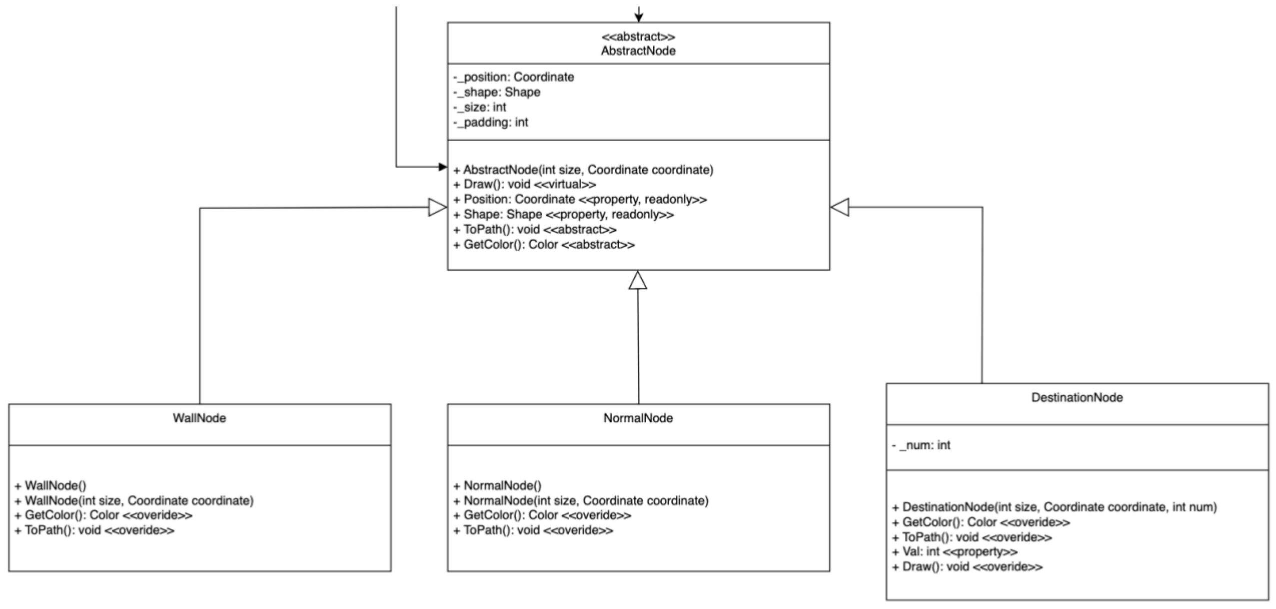

<<interface>>
INodeCollection
+ Reset(): void
+ Builder: IGraphBuilder<<property,readonly>>
+ CreateDepthFirstIterator(): NodeIterator
+ CreateBreadthFirstIterator(): NodeIterator
+ CreateAStarIterator(): NodeIterator
+ UpdateEvent(): void
Implements
Grid
- _builder: GraphBuilder
+ Grid(GraphBuilder builder)
+ Reset(): void
+ Builder: IGraphBuilder<<property,readonly>>
+ CreateDepthFirstIterator(): NodeIterator
+ CreateBreadthFirstIterator(): NodeIterator
+ CreateAStarIterator(): NodeIterator
+ UpdateEvent(): void
<<abstract>>
NodeIterator
- _grid: INodeCollection
-_pathTable: Dictionary<AbstractNode, AbstractNode>
-_visited: HashSet<AbstractNode>
-_destinations: List<AbstractNode>
+ NodeIterator(INodeCollection grid)
+ Destinations: List<AbstractNode> <<property>>
+ PathTable: Dictionary<AbstractNode,AbstractNode> <<readonly>>
+ AddToPathTable(AbstractNode key, AbstractNode val): void
+ CanVisit(AbstractNode): bool
+ Reset(): void <<virtual>>
+ Visited(): HashSet<AbstractNode> <<property,readonly>>
+ Highlight(AbstractNode node): void
+ AddNode(AbstractNode node): void <<abstract>>
+ HasNext(): bool <<abstract>>
+ NextNode(): AbstractNode <<abstract>>
DepthFirstIterator
-_st: Stack<AbstractNode>
+ AddNode(AbstractNode): void <<overide>>
+ HasNext(): bool <<overide>>
+ Reset(): void <<overide>>
+ NextNode(): AbstractNode <<overide>>
BreadthFirstIterator
-_q: Queue<AbstractNode>
+ AddNode(AbstractNode): void <<overide>>
+ HasNext(): bool <<overide>>
+ Reset(): void <<overide>>
+ NextNode(): AbstractNode <<overide>>
AStarIterator
-_openHeap: DistanceHeap
-_closeSet: HashSet<AbstractNode>
-_costTable: Dictionary<AbstractNode, double>
+ AddNode(AbstractNode): void <<overide>>
+ HasNext(): bool <<overide>>
+ Reset(): void <<overide>>
+ NextNode(): AbstractNode <<overide>>
- EuclideanDistance(AbstractNode node1, AbstractNode node2)
- HCost(AbstractNode node): double

Figure 10 Node Iterator

The terms of node iterator will control and return the next node to the GraphTraversal so the GraphTraversal could reply on it and complete the process of traverse. The Abstraction of Node Iterator implemented some necessary features of the concrete iterator so that concrete iterator class could adopt and make use of it. Additionally, The Node Collection is responsible for creating the Iterator.

Last but not least, each cell in the grid will be implemented by the node. There are three different kinds of node so that the traversal could control the node more easily. For example, Wall node is untouchable while the use of Normal node is specifically allowed to traverse.



<<abstract>>
AbstractNode
-_position: Coordinate
-_shape: Shape
-_size: int
-_padding: int
+ AbstractNode(int size, Coordinate coordinate)
+ Draw(): void <<virtual>>
+ Position: Coordinate <<property, readonly>>
+ Shape: Shape <<property, readonly>>
+ ToPath(): void <<abstract>>
+ GetColor(): Color <<abstract>>
WallNode
+ WallNode()
+ WallNode(int size, Coordinate coordinate)
+ GetColor(): Color <<overide>>
+ ToPath(): void <<overide>>
NormalNode
+ NormalNode()
+ NormalNode(int size, Coordinate coordinate)
+ GetColor(): Color <<overide>>
+ ToPath(): void <<overide>>
DestinationNode
- _num: int
+ DestinationNode(int size, Coordinate coordinate, int num)
+ GetColor(): Color <<overide>>
+ ToPath(): void <<overide>>
+ Val: int <<property>>
+ Draw(): void <<overide>>

Figure 11 Node

B. *Design Patterns*

The use of design patterns is very common in the design. Specifically, there are mainly four core design patterns which hve been used. In *Figure 7* and *Figure 8*, they are the adoption of Strategy pattern which allow the client as well as the button to have different strategy implementation and it could be share by different class in the architecture. The GraphBuilder class make use of the Builder pattern while the node collection will call different method of the builder to build it. The process of bulding the graph has been shown when users adding wall node. One of the core designs patterns implementations is Iterator pattern. The use of it will make the algorithm implementation behind each algorithm become abstract. The GraphTraversal only need to ask for the next node. Observer pattern has also been used. The process when the GraphTraversal find the path and it wants to ask the node to highlight itself become a path. Also, each individual click action of IOnClick interface is regarded as Observer pattern because it requires on subscription which is the client.

IV. Conclusion

In conclusion, the program has successfully followed the Object-Oriented principles as well as having implemented some of popular design patterns to enhance the reusability of code. The program has been inspired a lot from the original version by Clement Mihailescu on Github.