Architecture Design

Оглавление

[Architecture Design: Rules 3](#_Toc24238764)

[Good Architecture Rules: 3](#_Toc24238765)

[S.O.L.I.D. Rules 3](#_Toc24238766)

[Architecture Design: Decomposition 4](#_Toc24238767)

[Hierarchy 4](#_Toc24238768)

[Functionality 5](#_Toc24238769)

[GUI [View] 5](#_Toc24238770)

[Business Logic 6](#_Toc24238771)

[Agent and Map 9](#_Toc24238772)

[Algorithm Package 12](#_Toc24238773)

# Architecture Design: Rules

## Good Architecture Rules:

|  |  |  |
| --- | --- | --- |
| **Rule** | Description | Availability |
| **Scalability** | The ability to expand the system and increase its productivity by adding new modules. |  |
| **Maintainability** | Changing one module does not require changing other modules. |  |
| **Swappability** | The module is easy to replace with another. |  |
| **Unit Testing** | The module can be disconnected from all others and tested / repaired |  |
| **Reusability** | The module can be reused in other programs and other environments. |  |
| **Maintenance** | A module-based program is easier to understand and maintain. |  |

## S.O.L.I.D. Rules

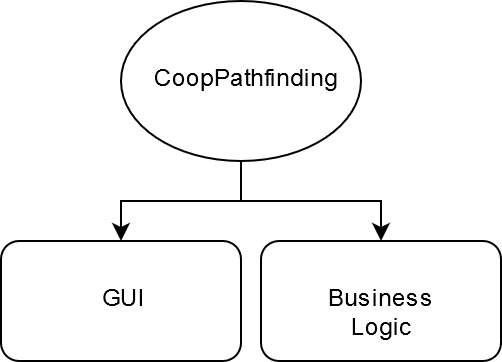
|  |  |  |
| --- | --- | --- |
| **Rule** | Description | Availability |
| Single responsibility principle | Class has one job to do. Each change in requirements can be done by changing just one class. |  |
| Open/closed principle | Class is happy (open) to be used by others. Class is not happy (closed) to be changed by others. |  |
| Liskov substitution principle | Class can be replaced by any of its children. Children classes inherit parent's behaviours. |  |
| Interface segregation principle | When classes promise each other something, they should separate these promises (interfaces) into many small promises, so it's easier to understand. |  |
| Dependency inversion principle | When classes talk to each other in a very specific way, they both depend on each other to never change. Instead classes should use promises (interfaces, parents), so classes can change as long as they keep the promise. |  |

# Architecture Design: Decomposition

## Hierarchy

Let’s use the MVC (Model – View – Controller) design pattern:

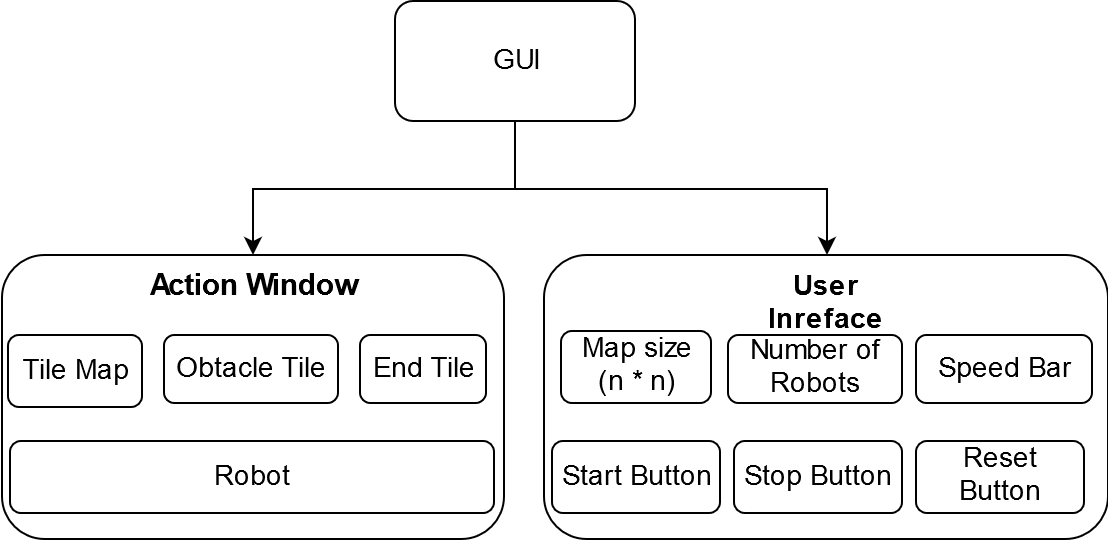
|  |  |  |
| --- | --- | --- |
| Part Number | Description | Part Name |
| 1 | It’s main logic of this application (business logic). | Model |
| 2 | An interface for this application. (Representation of information) | View |
| 3 | Accepts input and converts it to commands for the model or view | Controller |

1.  The ***model*** is responsible for managing the data of the application. It receives user input from the controller.
2. The ***view*** means presentation of the model in a particular format.
3. The ***controller*** responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

## Functionality

### GUI [View]

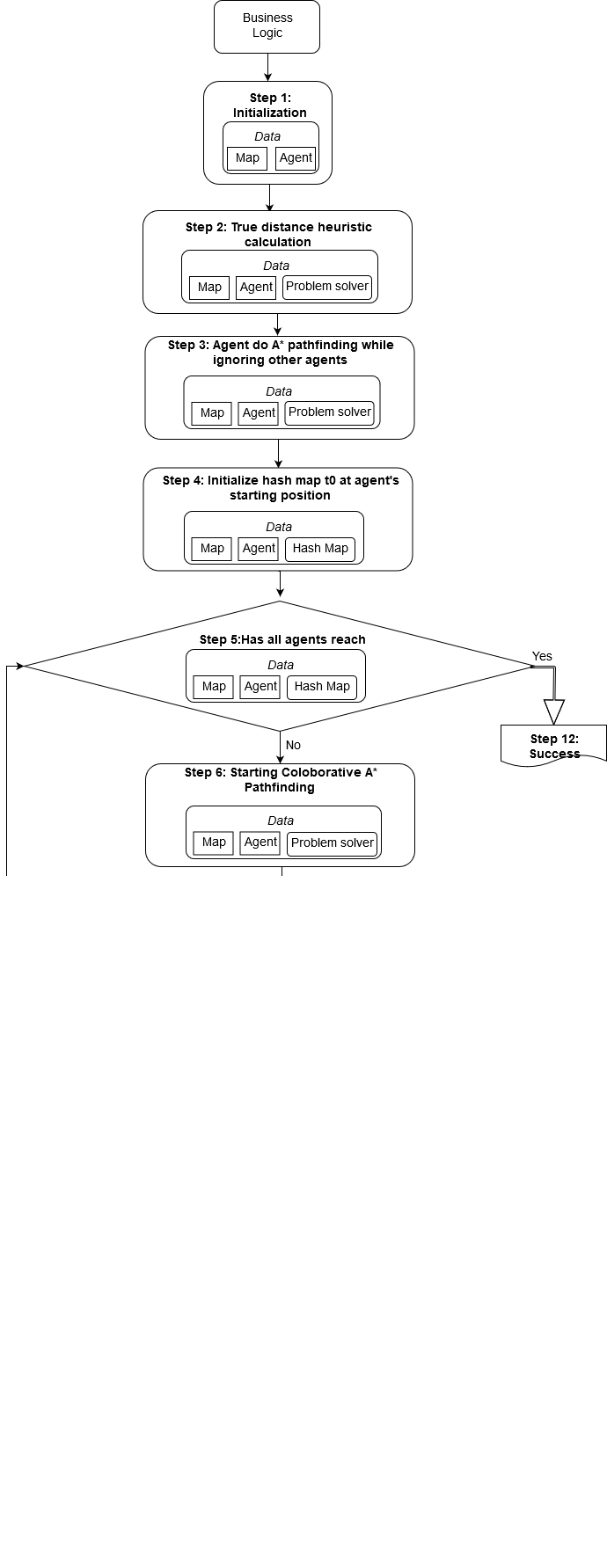
|  |  |  |  |
| --- | --- | --- | --- |
| Submodule name | Object name | Description | Importance |
| Action Window | Tile Map | This is a map N\*N which consists of square tiles. Basic tile which is clear (neither Obstacle nor Finish) **is white**. | Important |
| Obstacle Tile | This is Obstacle Tile. Walls for the Robot (If Robots go on the black tile then he crashing). The color is black. | Important |
| End Tile | The End Tile is a finish for the each Robot  These tiles may or may not be the same. The color of this tile is the same as Robot color. If several number of Robots have the same finish than tile ??? | Important |
| Robot | Robot it’s a cyrcle with their own number and probably color. | Important |
| User Interface | Map size | List with size N\*N.  [10\*10; 20\*20; 30\*30 …] | Important |
| Number of Robots | Input field where user set the number of Robots. Number of the Robots should be in the range [1; N\*N] where N is size (row/column)  16 | Important |
| Speed Bar | This is bar which represent velovity of animation. | Not Important |
| Start Button | Starting animation | Important |
| Reset Button | Reclaim start positions | Important |
| Stop Button | Stop animation | Important |

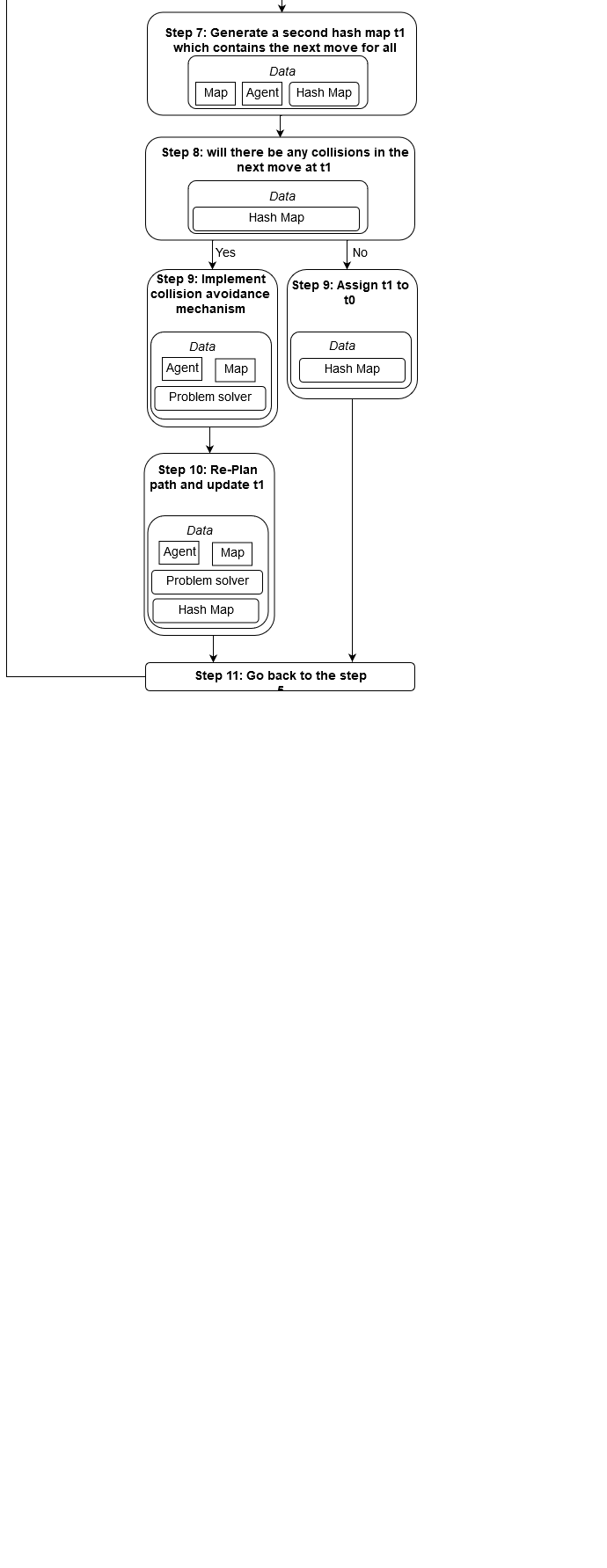


### Business Logic

|  |  |  |
| --- | --- | --- |
| Data Name | Description | Type |
| Agent | This is the Robot. I will Describe this class a little bit later. | Class |
| Map | The Map-class. =\ | Class |
| Problem Solver | This is the package which contain the algorithms for the solving. | Package (module) |
| Hash Map | Standard library package | STD Package |

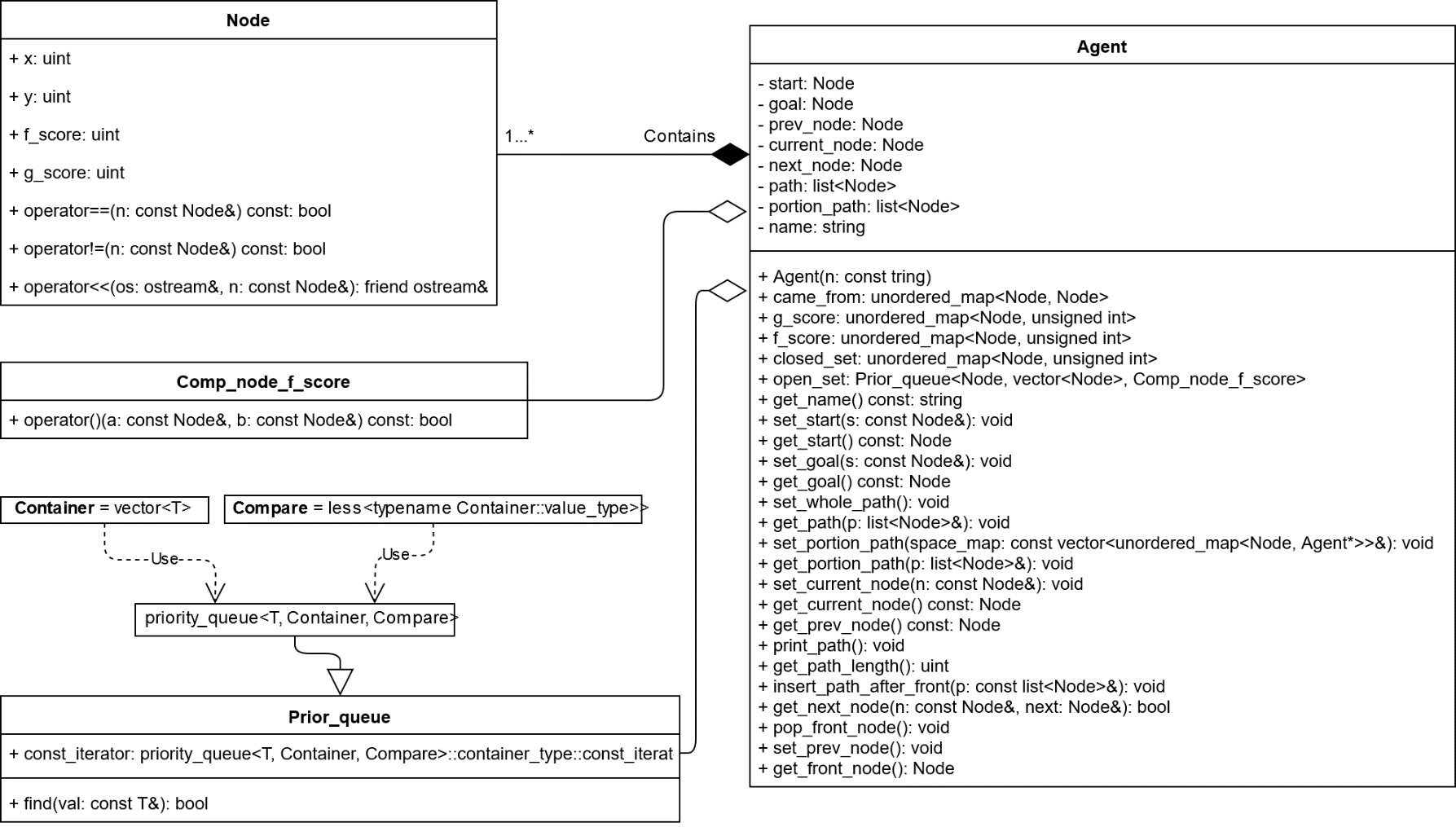
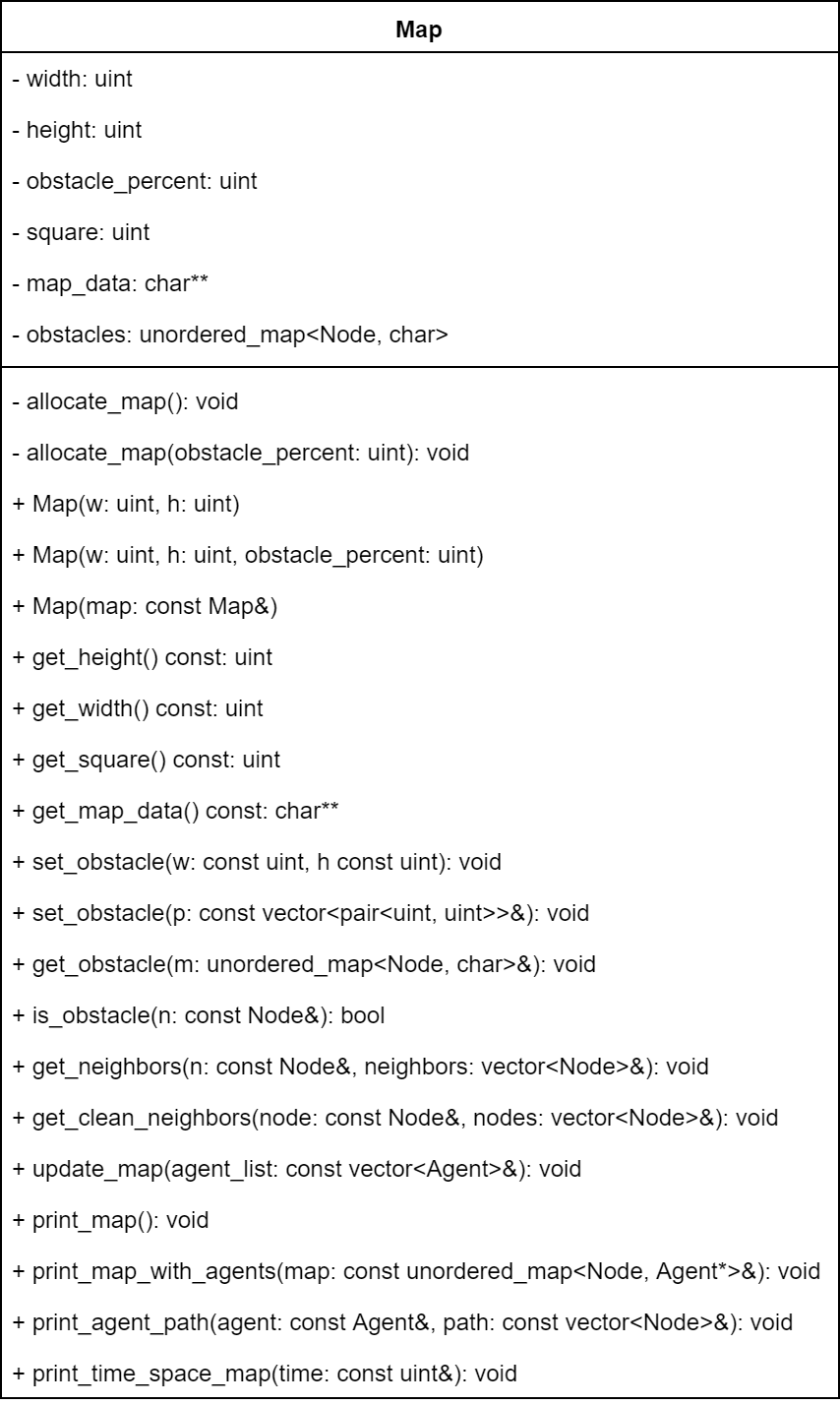
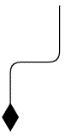
Overall objective is collaboratively pathfinding for all agents, make sure they don’t collide, and reach goal safe and sound. System initiates when all agents are deployed at their own starting position on the map. System shall end when all agents reach goal position or there is no pathfinding solution. System returns an error message when agents cannot find path.





### Agent and Map

|  |  |  |
| --- | --- | --- |
| Name | Description | Type |
| Node | The basic unit of a path of an agent, include x, y coordinates, f\_score, g\_score. | Class (type) |
| Agent | An object with a name (number, of aggent), starting point and goal point. This object will move across the map, from the start point toward the goal point. | Class (type) |
| (x, y) | Point, basic unit of space map | Uint |
| g\_score | The g(x) function, which estimate the distance travelled by agent so far from start to Node. | Uint |
| f\_score | The f(x) = g(x) + heuristic distance (Manhattan Distance Heuristic), meaning the estimation of distance from start to goal. Implemented in hash table <Node, f\_score> | Uint |
| closed\_set | Explored points (Nodes), meaning its parent, g\_score, f\_score is calculated | Unordered\_map |
| open\_set | The set of points (Nodes) newly found, but unexplored. | Prior\_queue |
| came\_from | A hash table <child, parent> indicates that child node came from parent node. | Unordered\_map |
| space\_map | A two – dimensional grid map consists X and Y coordinates. | Unordered\_map |
| name | The name, namely number – string | String (method) |
| get\_name | Returns the name of an Agent | String (method) |
| set\_start | Set starting position | Void (method) |
| get\_start | Returns starting position | Node (method) |
| set\_goal | Set goal position | Void (method) |
| get\_goal | Returns goal position | Node (method) |
| set\_whole\_path | Using A\* Search to set complete | Void (method) |
| get\_path | Get agent path | Void (method) |
| set\_portion\_path | Set agent portion path, i.e. each agent walks n steps in turn, where n is the window size. | Void (method) |
| get\_portion\_path | Get agent portion path | Void (method) |
| set\_current\_node | Set agent current position | Void (method) |
| get\_current\_node | Get agent current position | Node (method) |
| set\_prev\_node | Set agent previous position | Void (method) |
| get\_prev\_node | Get agent previous position | Node (method) |
| get\_next\_node | Get agent next position, return false if agent is at end of window | Bool (method) |
| print\_path | Print agent’s whole path using basic A\* Search | Void (method) |
| get\_path\_length | Get agent path length | Uint (method) |
| insert\_path\_after\_front | Insert a path after current node | Void (method) |
| pop\_front\_node | Pop out current node from path | Void (method) |
| get\_front\_node | Took at agent current position | Void (method) |



### Problem – Solver Package