

12.11.2019

Architecture Design



Aleksandr Kasian; Oskar Boer

Table of contents

Architecture Design: Rules.....	2
Good Architecture Rules:	2
S.O.L.I.D. Rules	2
Architecture Design: Decomposition.....	3
Hierarchy.....	3
Functionality	4
GUI [View]	4
Business Logic.....	5
Agent and Map	8
Problem – Solver Package	11
Conclusion	11
Bibliography	11

Architecture Design: Rules

Good Architecture Rules:

Rule	Description	Availability
Scalability	The ability to expand the system and increase its productivity by adding new modules.	Yes
Maintainability	Changing one module does not require changing other modules.	No
Swappability	The module is easy to replace with another.	No
Unit Testing	The module can be disconnected from all others and tested / repaired	No
Reusability	The module can be reused in other programs and other environments.	Yes
Maintenance	A module-based program is easier to understand and maintain.	Yes

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.	Yes
Open/closed principle	Class is happy (open) to be used by others. Class is not happy (closed) to be changed by others.	No
Liskov substitution principle	Class can be replaced by any of its children. Children classes inherit parent's behaviours.	Yes
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.	Yes
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.	Yes

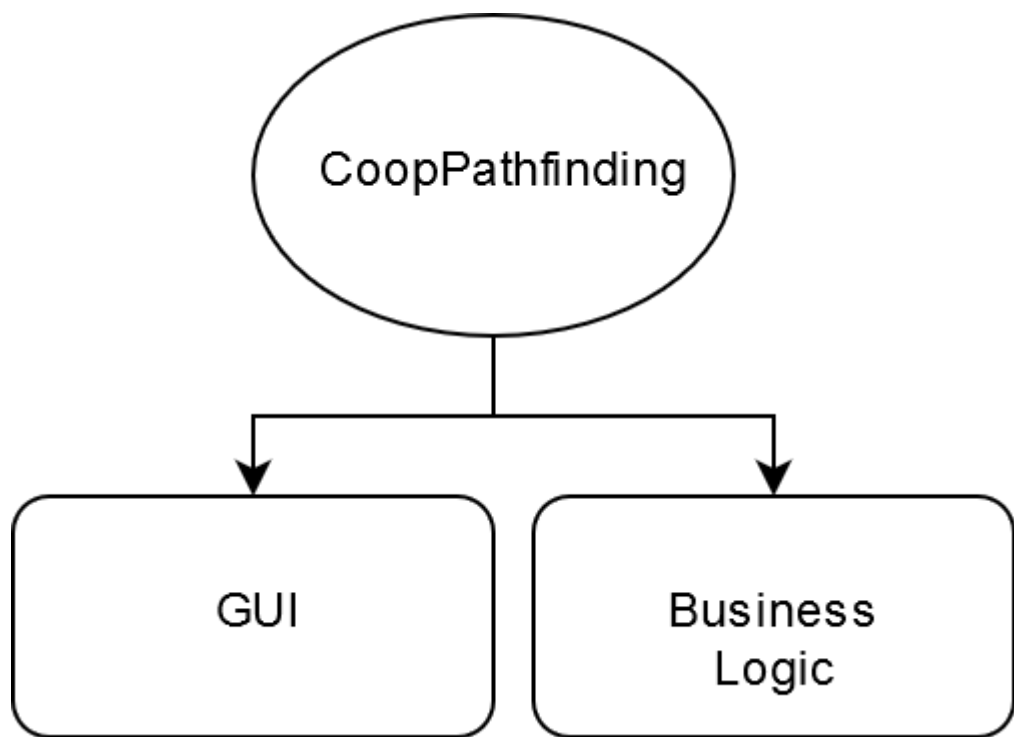
Architecture Design: Decomposition

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

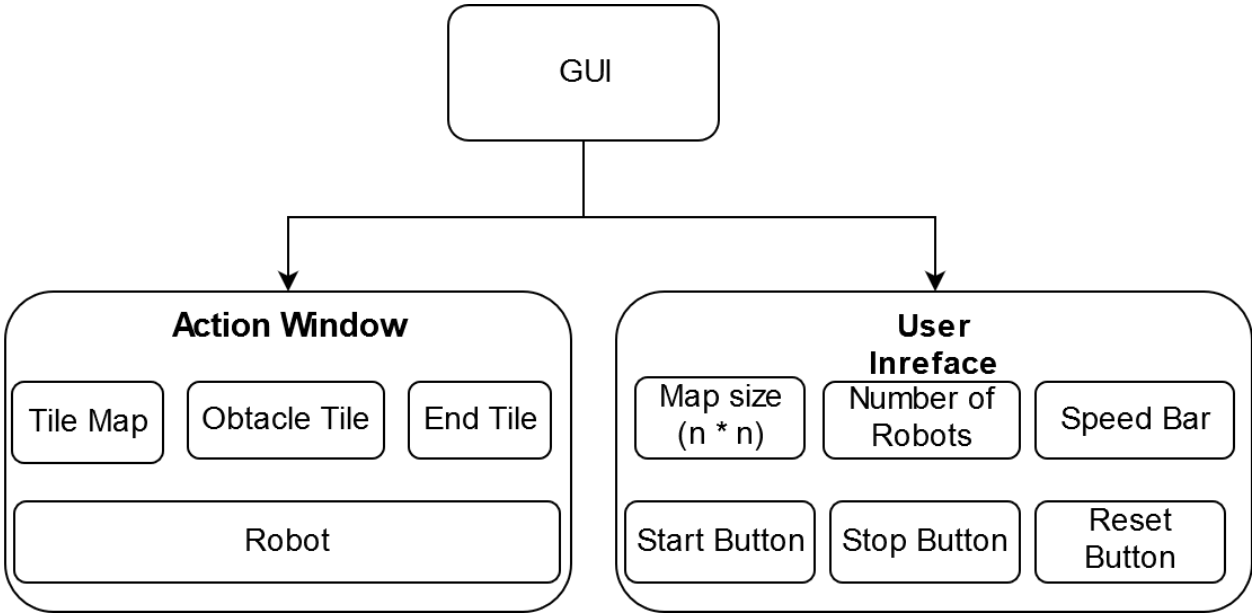
- 1) The **model** is responsible for managing the data of the application. It receives user input from the controller.
- 2) The **view** means presentation of the model in a particular format.
- 3) The **controller** responds to the user input and performs interactions on the data model objects. The 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 W*H 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 name and probably color. 	Important
User Interface	Map size	List with size W*H.	Important
	Number of Robots	Input field where user set the number of Robots. Number of the Robots should be in the range [1; N*M] where N/M is size (row/column)	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



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

*“To demonstrate the main purpose, consider Figure 1. Initially, agents A and B (agents locations are marked with white circles), each need to cross the entire grid to get to their goals while agent C is one step away from its goal and the reservation table is empty. The first **plan-move** cycle begins and each of the agents find a path to their goal (Figure 1(a)). By simulating the paths, the first conflict is found between the paths of agents A and B at time point t (marked with a red X). In this example, agent B is chosen arbitrarily as the **conflict master**. We assume a fixed window size of size W and that the reservation window should be positioned evenly around the conflict. Thus, we select w_{start} and w_{end} such that W st-points around the conflict are reserved by agent B (Figure 1(b)). Next, the move phase begins (Figure 1(c)) and the agents proceed to their locations at w_{start} . A new plan phase begins at step (d). Since agent B reserved the interval adjacent to conflict c , agent A plans a detour - a path that conflicts with agent C. Now, C gets to reserve the interval (Step (e)) and the next move phase advances agent A near the location of c (Figure 1(f)). Next, a new plan is found and all agents. Agent C again reserves the interval around the conflict and agent A plans an alternative path to its destination. This time, none of the agents have conflicts. The space requirements of the reservation table used in WHCA* and HCA* depends on the number of agents ($W \cdot |A|$ in WHCA*) since all agents reserve st-points at each phase. In CO-WHCA*, the reservation table is usually a vector of size W since most of the time, only one agent uses the reservation table for a limited duration. One exception to this is when multiple conflicts occur at the same time.”*

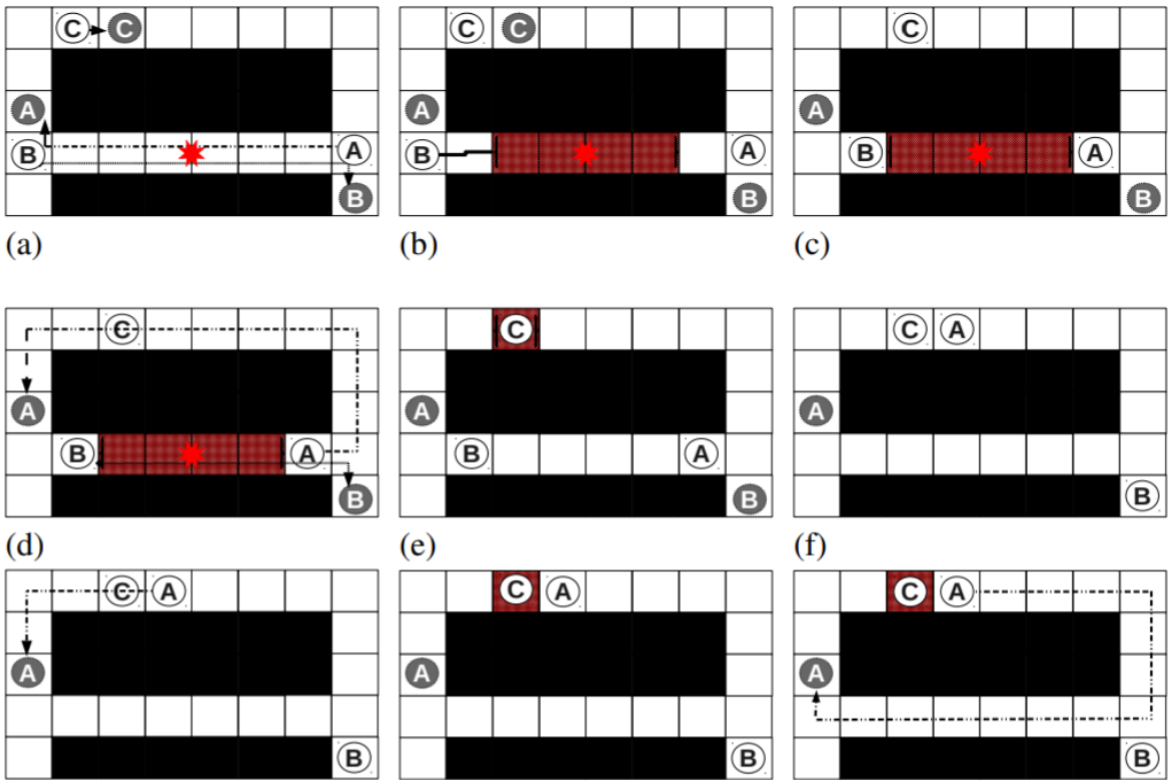
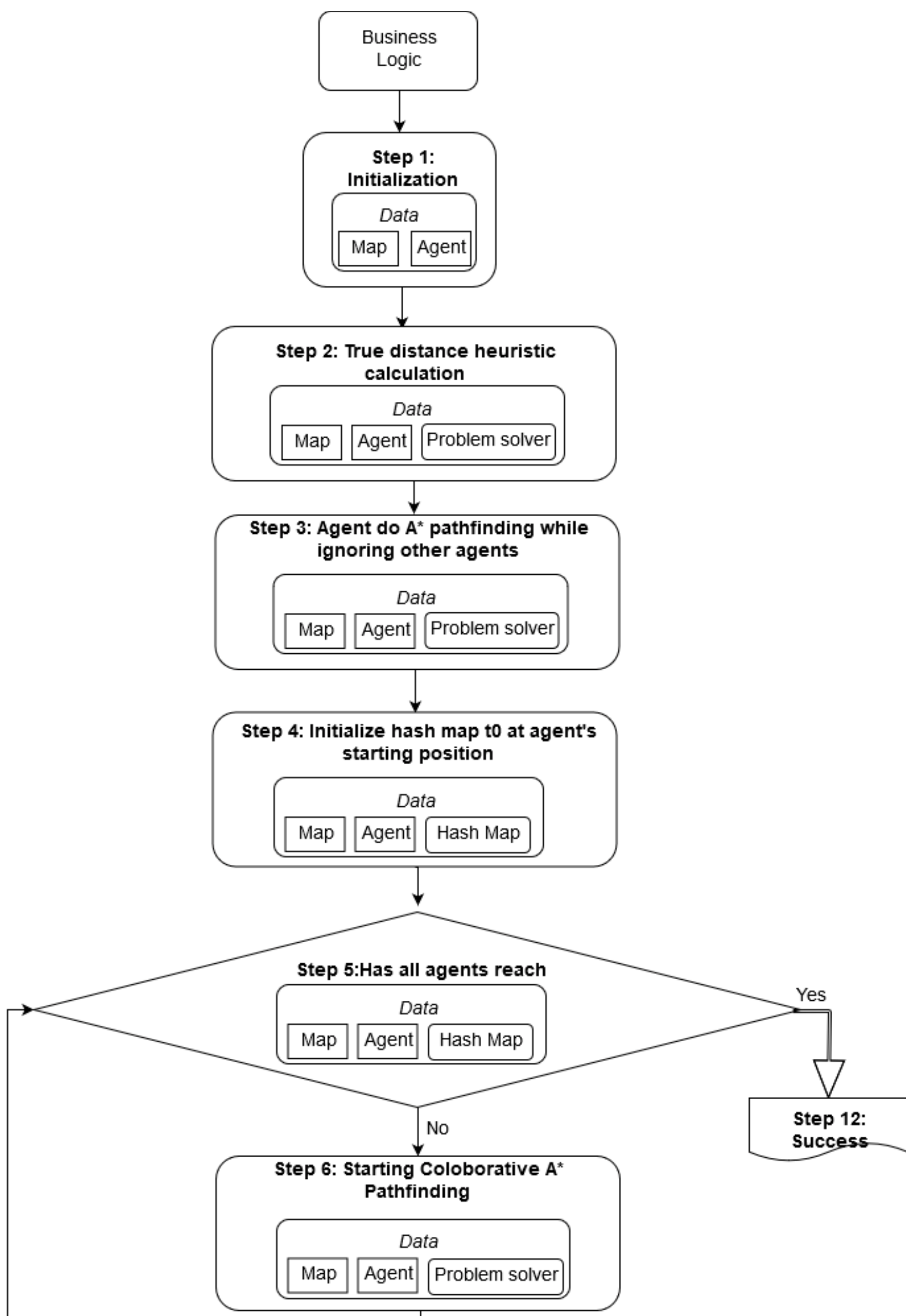
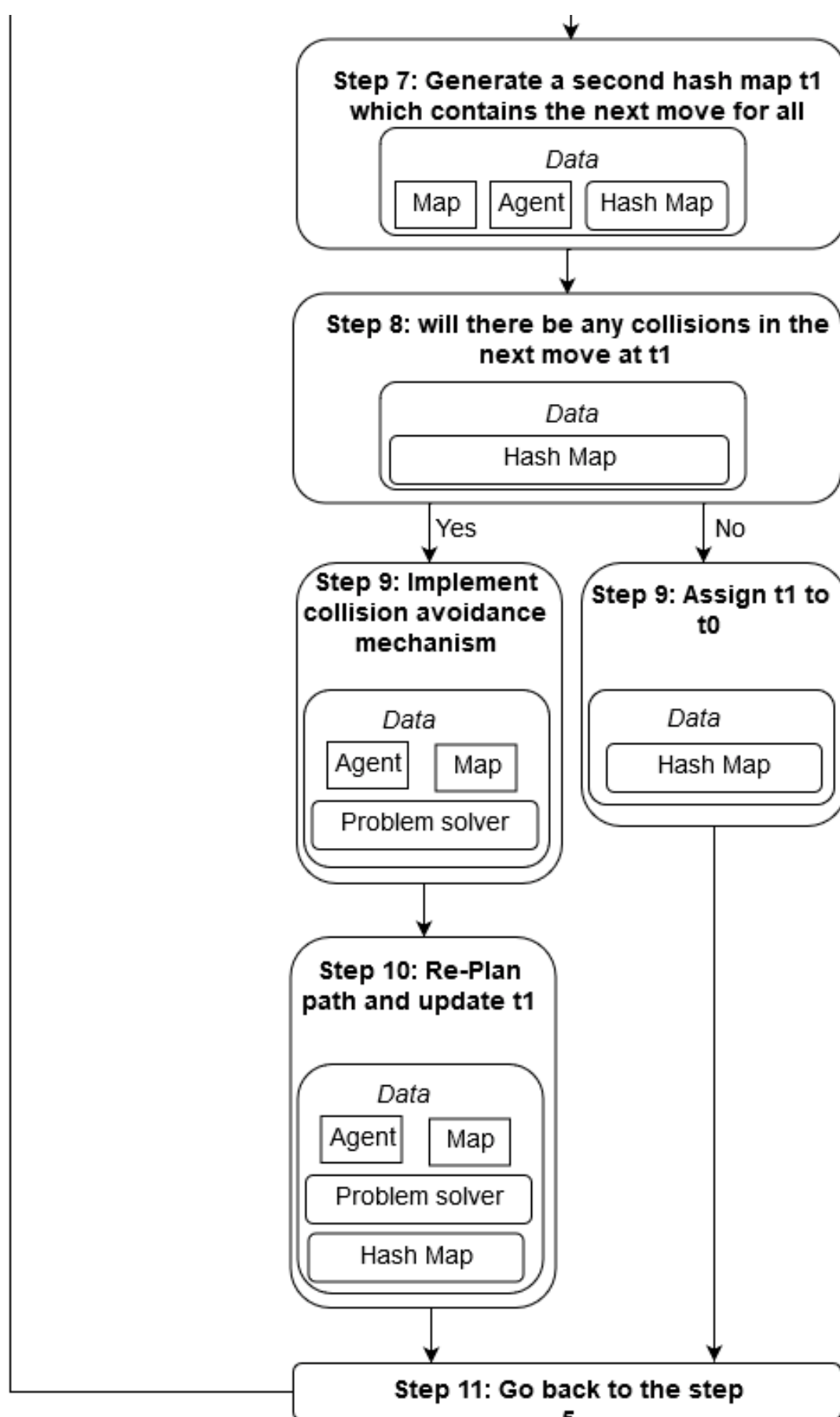


Figure 1: WHCA* Running





Agent

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)

Map

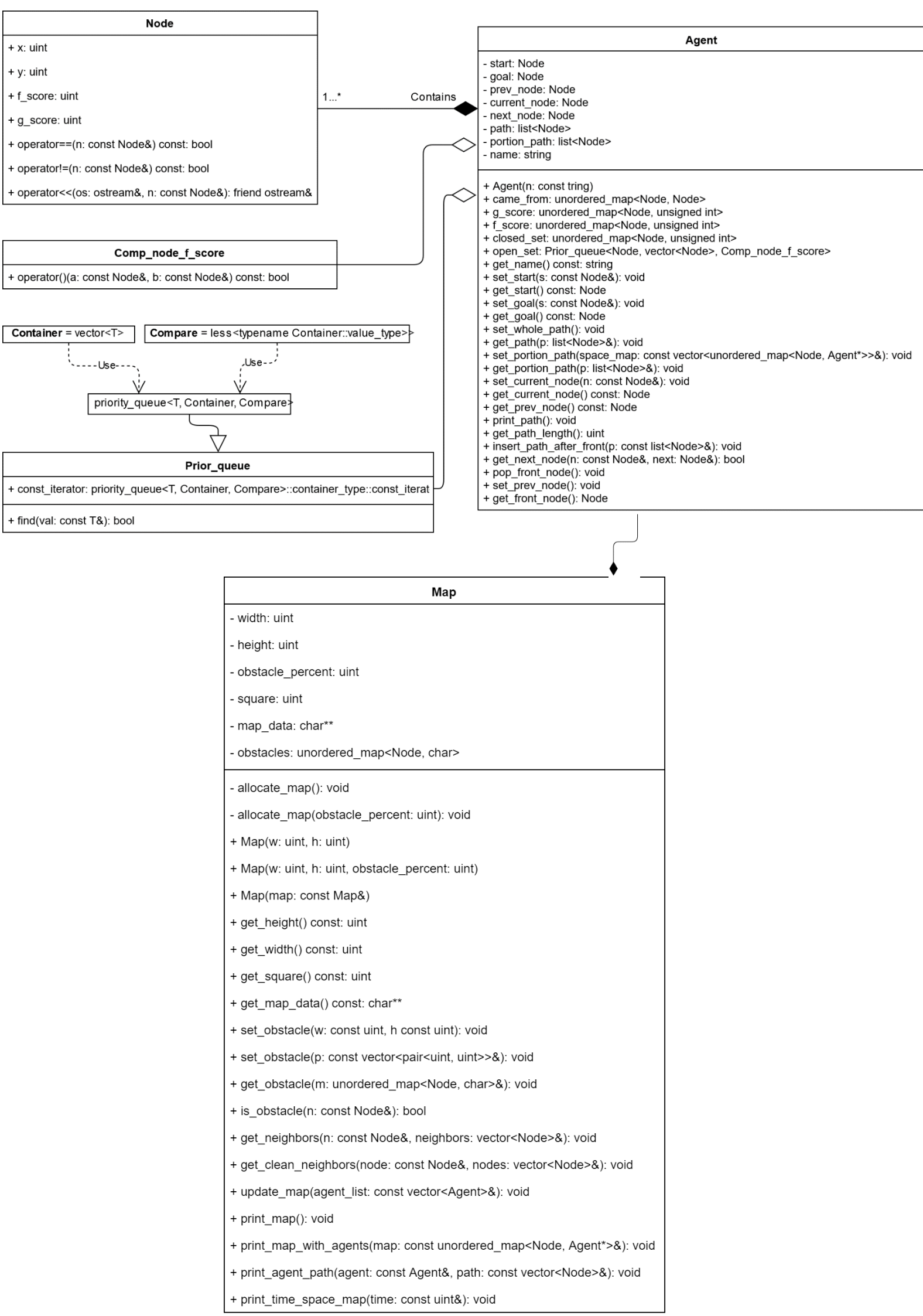
Name	Description	Type
Map	The map where the agents will find the paths.	Class (type)
(w, h)	Width and height of the map.	uint
obstacle_percent	The percent of obstacles relative to the entire map.	uint
square	The square of the map, w*h.	uint
get_height	Get height of the map	uint (method)
get_width	Get width of the map	uint (method)
get_square	Get square (W*H) of the map	uint (method)
get_map_data	Get all data of the map. (include start/end point, agents, obstacles e.t.c.)	char** (method)
set_obstacle	Set obstacle on the map	void (method)
get_obstacle	Get obstacle	void (method)
is_obstacle	Is a node as an obstacle	bool (method)
get_neighbors	Get neighbors and we don't care either node is an obstacle or node is empty or it's an agent e.t.c.	void (method)
get_clean_neighbors	Get neighbors which are not contains an agent.	void (method)
update_map	Update the map	void (method)
print_map	Just print a map	void (method)
print_map_with_agents	Print a map with agents	void (method)
print_agent_path	Print agent path.	void (method)
print_time_space_map	Print time space map	void (method)

To seek high performance, Agent architecture is meant be designed as simple as possible.

On the contrary, the coordination mechanism in terms of solving interdependency issues shall be complex enough to consider all the scenarios. We want agents cooperate only when they have to. In other words, agents do not need to communicate with others if they know in advance they will not collide. Collison detection is the coordinator’s job. Once coordinator finish planning the path for all the agents, agents can walk without worry about collision.

The coordination mechanism is embedded inside each agent (Probably I will separate the coordination from each agent). From a single agent perspective, it has a name, a starting point, a goal point. The knowledge of each agent includes map, closed set, open set, g-score hash map, f-score hash map.

Most of coordination, interaction are embedded inside problem – solver functions. A star search, Reverse Resumable A Star Search, True distance calculation, Collision detection and avoidance are separated from agent architecture. I will describe them each in detail in the “Project-Explanation” document.



Name	Description	Type
Manhattan Distance Heuristic	An estimated distance ignoring obstacles between two points.	uint
True Distance Heuristic	It distance taking into account of obstacles.	uint
A*Search	Simple A*Search. Pretty good description tou will find on the Wikipedia by following the link	bool
Reverse Resumable A* Search	Back to the problem of calculating true distance heuristic, the solution is to use Reverse Resumable A* proposed by David Silver. But I will make the only difference, I will make a full search once, without resume. I plan to do it on GPU, because this method to expensive.	bool
Collision Detection And Avoidance	Agents interact through the space_time_map windowed time (W steps) in turn based on other agents on that space_time_map.	Several types

Conclusion

First of all, it is not the end. In future I will add some features such as “Path Rejoining”, “Predictor” (I described these in “Project-Explanation” document), agent – velocity, diagonal – movement, bad agents (whose purpose is to stop other agents), and so on, and so on.

Secondly, pathfinding is indeed very interesting project, and complex to solve if other constraints are considered. The project is meant to be a learning experience for myself. It is far from completing, and the current research trend on multi-agent pathfinding is far ahead from this little project.

Bibliography

[1] Silver D. (2005). Cooperative Pathfinding. ([link](#))

[2] Zahy Bnaya and Ariel Felner. Conflict-Oriented Windowed Hierarchical Cooperative A*. ([link](#))