ECE9607: Collaborative A-star Pathfinding

Proposed Solution & Design Specification Report

Yan Ge

Department of Electrical & Computer Engineering
The University of Western Ontario

Date: 15, July 2017

Table of Content

2 Overview

In this document, we first describe the project goal and objectives, explain a few terminologies. Then we analysis the problem based from the perspective of CDS. After that we explain our proposed solution in section 3 which includes functional aspect and system architecture. In section 4, we elaborate the details of architecture consisting 4 layers, namely smart-object based, problem-solver component, coordination component, and finally communication component. In next section, we will point out some of main challenges and implementation constraints. Finally, we draw conclusion and future expectation.

2.1 Project Description: Goal and Objectives

The main goal is to allow agents collaboratively to path find using A* search and A* alike algorithm.

The following lists some of current sub-goals in terms of CDS and SO based paradigm.

- From single-agent perspective,
    - A SO is fully capable of performing basic A* pathfinding with the knowledge of map, starting position, goal position autonomously regardless of the structure of open environment. Specifically, given any map, any start or goal position, agent can correctly identify a path or no path without knowledge of other SO.
    - A SO can communicate with central coordinator directly, so that coordinator can effect another SO or a group of SOes in terms of solving conflict, achieving the minimum overall cost of all agents.
    - A SO can share its knowledge with other SOes indirectly through central coordinator such that the information will allow other SOes to replan path, avoid collision in advance.
    - A SO must obey the coordinator.
- From multi-agent perspective,
    - Shared resource interdependency, such as map information, all SOes' existing or future path.
    - Interest interdependency, in terms of conflicting interest, collision detection and avoidance.
    - Priority interdependency, in terms of agents' priority assignment, which agent plan path first, and how to achieve the optimal routes with minimum cost.
    - Autonomous coordination, in terms of pattern of communication at collision points, and associated mechanism in terms of resolving conflict, assigning priority and replanning path for agents.
    - Distributed problem solving, meaning each agent proposes its route greedily regardless of other agents' plan, and submits its plan to coordinator to generate a global plan.

2.2 Terminology & Definitions

Provide a list of all used terminology and acronyms with brief definition for each

- A* search — An algorithm for finding shortest path from start to destination.
- Reverse Resumable A* search

  Run A* search backwards, set it original start as goal, goal as start.

  The result gScore is the true distance to the goal taking account of obstacles.
- space map — A two-dimensional grid map consists of X and Y coordinates.
- (x, y) — Point, basic unit of space map.

- Space-time map Add time dimension to space map.
- (x, y, t) point at time t, e.g (x, y, 0) is the starting position (x, y) at time 0.
- Agent an object with a name, starting point and goal point.
- Node basic unit of a path of an agent, include x, y coordinates, fScore, gScore.
- Path Agent's path from start to goal
- 'X' Obstacle region
- '0' Walkable region.
- Manhattan Distance Heuristic
  An estimated distance ignoring obstacles between two points.
- Admissible heuristic
  The distance heuristic never overestimates the distance to the goal.
  Meaning it is the least distance one could possible get.
- True Distance Heuristic
  The true distance taking account of obstacles to the goal.
- Neighbor adjacent points of (x, y), e.g. (x-1, y), (x+1, y), (x, y-1), (x, y+1), (diagonal excluded)
- cameFrom a hash table <child, parent> indicating child node came from parent node.
- gScore The distance travelled by agent so far from start to Node, implement as a hash table <Node, gScore>
- fScore fScore = gScore + Manhattan Distance Heuristic, meaning the estimation of distance from start to goal. Also implemented in hash table <Node, fScore>
- ClosedSet Explored points, meaning its parent, gScore, fScore is calculated.
- OpenSet a set of points newly found, but unexplored.
- Collision Two or more agents at one Node or point at same time.

3 Problem Analysis

This section describes analysis in detail regarding application problem analysis and CDS based issues.

3.1 Application Problem Analysis

From the perspective of Smart-object, this application allows one single smart object or agent to find the best path with shortest distance. Once a space map is given, with only one agent walking on the map, given one specific start and goal location, agent will find path using basic A-star search algorithm.

However, multi-agent pathfinding is not going to work with only the view from each individual smart-object. In other words, coordination is much needed from the cooperative distributed system which includes autonomously manage interdependency issues which includes

- knowledge, such as recognizing collision between agents and how to avoid them,
- resource sharing such as map and paths from other agents,
- conflicting interesting such as resolving collision and assign priorities to agents,
- and last but not at least, the autonomous coordination scheme which gives pattern of communication at a specific decision point, and mechanism to resolve issues.

3.2 CDS based Issues Analysis

Now we mainly focus on interdependency issues as illustrated above.

1. Capability interdependency
    a. In the case of multi-agent pathfinding, instead of planning whole path, i.e. from start all the way to goal, we used windowed approach while will be explained in next section.
    b. This windowed approach is essentially a decomposition of goals into sub-goals. Within a window, agent's sub-goal is to walk full windows size, agent may walk towards its final goal, may detour to coordinate, and cooperatively find sub-optimal path.
2. Interest interdependency
    a. In this project, we focus on resolving conflict, collision detection in terms of agent's path, priority, true distance to goal. As number of agents becomes large, the conflict interest interdependency issue is going to be exponentially difficult. Therefor we introduce decoupled approach.
    b. This decoupled approach is essentially divide map into many pieces of region of interest. Regions will overlap because one agent may pass from one region into another. The size of region and degree of overlapping determines the cooperative level in this project.
3. Knowledge interdependency
    a. Of course, map is a shared knowledge. Each agent has complete view of the whole map. Because each agent must know true distance heuristic prior path planning.
    b. Second, in terms of resolving conflict interest, some agents will have knowledge of other agents who has been on its path or will be soon.
    c. The level of coordination, in terms of sharing agents' route. E.g. agent A does not have to know agent B who is far away from her even though agent B is on agent A's planned path. We don't know for sure after 100 steps, they will meet with each other.
    d. Therefore, we propose knowledge is shared locally, not globally to improve efficiency of path planning.

4 Proposed Solution & Architecture

In this section, we first discuss some of functionalities, then how to evaluate our application in terms of quality factors, and finally, the system architecture in brief.

4.1 Functional Descriptions

The main function in this application is that with enough map information, agents can coordinate to find path from start to goal collaboratively.

In terms of real world application, one could easily adapt this application to real-world scenarios, including but not limited:

1. Robot search rescue in case such as earthquake
2. Self-driving cars route planning, car parking, etc.
3. Warehouse robot path planning, this is like case 1, but different in terms of open environment.
4. Aerial vehicle route path planning, in this case, open environment is in 3D.
5. Computer game, e.g. soccer game is still an open research topic.

4.2 Quality Factors Descriptions

In terms of qualifying our application, we list some of the factors that we would like to consider:

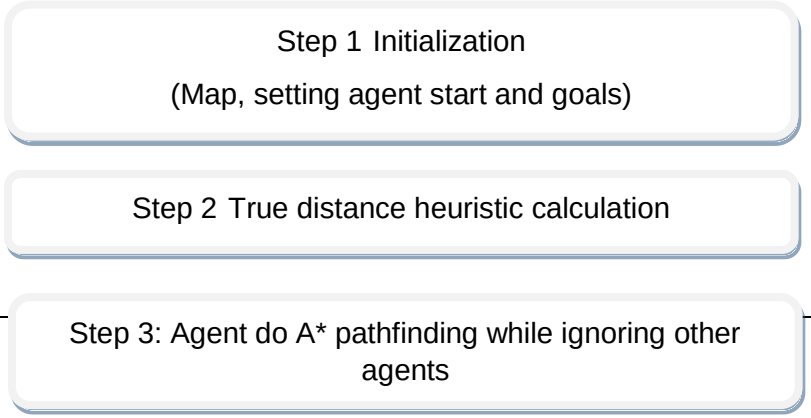
1. Number of agents succeed in pathfinding, number of agents failed vice versa.
2. Number of collision avoided, collision detected.
3. (hardest) Finding the optimal cost of path of all agents, taking account of everything, even if all agents avoid each other and reach goal, is this the best path planning in terms of time and cost?
4. Randomness. Meaning given a map with random obstacles, random agents' start and goal.
5. The autonomy of problem solving in terms of path planning, conflict resolving.
6. The degree of cooperation, the level of autonomy, e.g. if agent unable to path finding, should it wait and for how long, and to search how far.
7. The time and cost spent during communication. Meaning how long it takes during path planning before agents are ready to move.

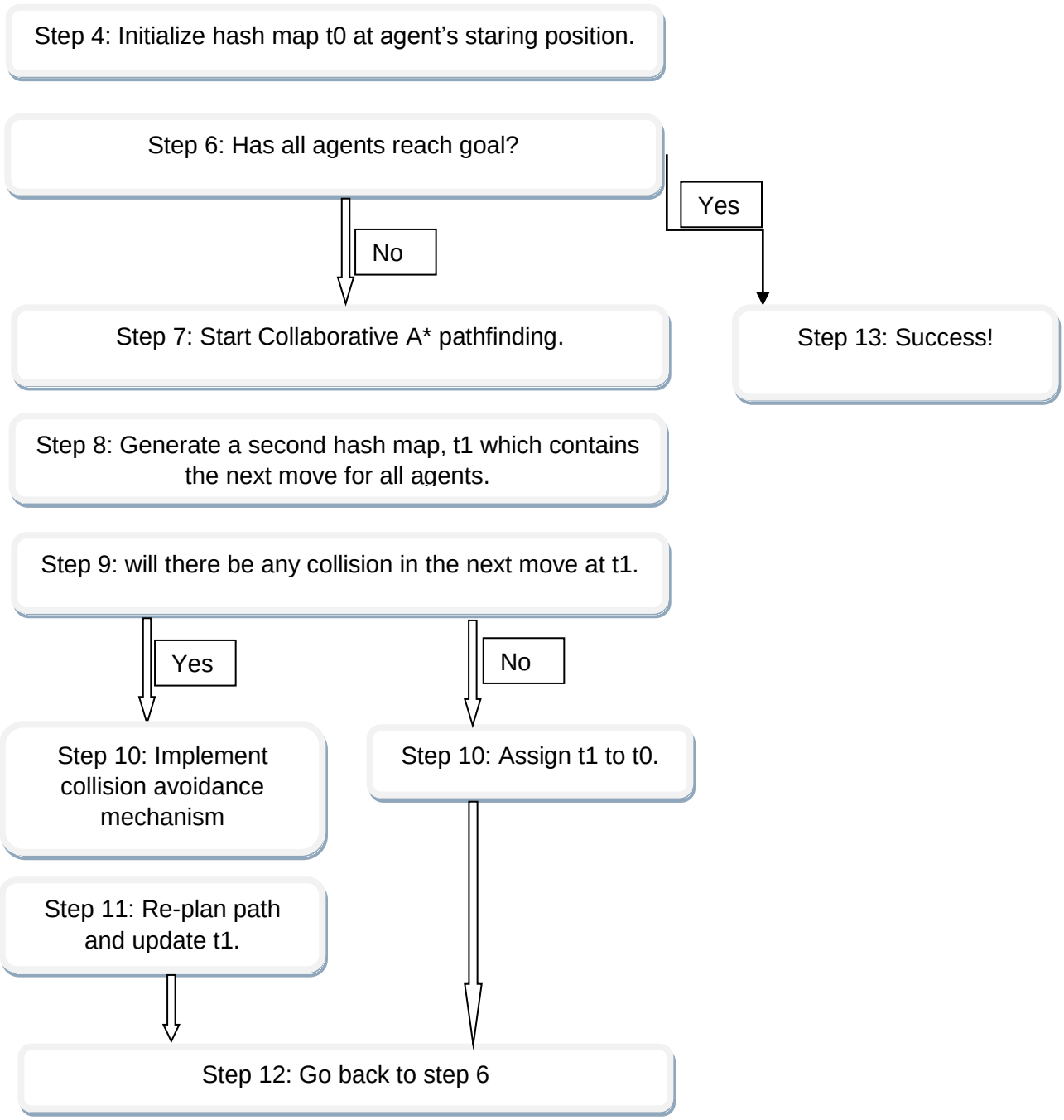
4.3 System Architecture

In the following sections, we will describe logical architecture. We then discuss each in detail in section 5.

4.3.1 Logical Architecture

The following layered diagram show logical architecture.




Step 4: Initialize hash map t0 at agent's staring position.
Step 6: Has all agents reach goal?
Yes
No
Step 7: Start Collaborative A* pathfinding.
Step 13: Success!
Step 8: Generate a second hash map, t1 which contains the next move for all agents.
Step 9: will there be any collision in the next move at t1.
Yes
No
Step 10: Implement collision avoidance mechanism
Step 10: Assign t1 to t0.
Step 11: Re-plan path and update t1.
Step 12: Go back to step 6

5 Detailed Architecture

A detailed description of the system's logical architecture for the application is presented within the subsections below.

5.1 Smart-Object based Architecture

The smart-object is designed as Agent class. It has private members including agent's name, start and goal, current position, next move in the path, and whole path from start to the end.

It has several public functions including set and get private members, modifying method to change current path, and finally, hash map store the agent's knowledge regarding gScore, fScore, cameFrom, closedSet. We also implement binary heap (priority_queue) to the openSet for better performance.

```
/*********************************************************************/
// Agent class.
class Agent
{
public:
	Agent(char c) : name(c) { path.clear(); };

	bool operator==(const Agent &a) const {
		return (name == a.getName());
	}

	unordered_map<Node, Node> cameFrom; // <child, parent>
	unordered_map<Node, unsigned int> gScore;
	unordered_map<Node, unsigned int> fScore;
	unordered_map<Node, unsigned int> closedSet;  // calculated fscore nodes
	prioriyQueue<Node, std::vector<Node>, comp_node_fScore> openSet; // lowest fscore
on top()

	char getName() const { return name; };

	void setStart(const Node& s) { start = s; path.push_back(s); current_node = s; };
	Node getStart() const { return start; };

	void setGoal(const Node& g) { goal = g; };
	Node getGoal() const { return goal; };

	void setPath();
	void getPath(list<Node> &p);

	uint getTime() const { return time; };
	void setTime(uint t) { time = t; };

	Node set_current_node(const Node& n) { current_node = n; };
	Node get_current_node() const { return current_node; };

	void print_path();
	uint get_path_length() const { return path.size(); };
	void insert_path_to_front(const list<Node>& p);

	bool getNextNode(const Node& n, Node& next);
	bool getPrevNode(const Node& n, Node& prev);

	void pop_front_node_from_path() { if (!path.empty()) path.pop_front(); };
	Node get_front_node_from_path() { return path.empty() ? current_node :
path.front(); };

private:
	char name;
	uint time;
	Node start;
	Node goal;
	Node current_node;
	Node next_node;
	list<Node> path;
};
```

5.2 Problem-Solver Component Architecture

In terms of problem-solver component architecture, basically we implement several helper methods to Agent's capability. The following code shows the detailed implementation, including Manhattan distance heuristic, true distance heuristic, A*, reverse resumable A*. Of course, we will implement more methods to increase agent's problem-solving capabilities.

```
/*****************************************************************/
// manhattan distance heuristic.
inline uint manhattan_heuristic(const Node& a, const Node& b)
{// the cost moving form one spot to an adjacent spot vertically or horizontally is 10
	return 10 * (std::abs(static_cast<int>(a.x - b.x)) + std::abs(static_cast<int>(a.y
- b.y)));
}

/*****************************************************************/
// A-Star pathfinding for single agent.
bool aStar(Agent &agent, Map &map)
{
	// empty agent's closedSet
	if (!agent.closedSet.empty())
		agent.closedSet.clear();

	if (!agent.openSet.empty())
		cout << "openSet is not empty, error\n";

	Node start = agent.getStart();
	Node goal = agent.getGoal();

	// Initially, only the start node is known.
	agent.openSet.push(start);
	// The cost of going from start to start is zero.
	agent.gScore.insert({ start, 0 });
	// For the first node, that value is completely heuristic.
	agent.fScore.insert({ start, manhattan_heuristic(start, goal) });

	while (!agent.openSet.empty())
	{
		Node current = agent.openSet.top();
		agent.openSet.pop();

		if (current == goal)
			return true;

		agent.closedSet.insert({ current, agent.fScore[current] }); // change
closedSet to vector.
		vector<Node> Neighbors;
		map.getNeighbors(current, Neighbors);

		for (Node& neighbor : Neighbors) {
			// insert new nodes
			if (agent.cameFrom.find(neighbor) == agent.cameFrom.end()) {
				agent.cameFrom.insert(make_pair(neighbor, current));
				agent.gScore.insert(make_pair(neighbor, MAX));
				agent.fScore.insert(make_pair(neighbor, MAX));
			}
			if (agent.closedSet.find(neighbor) != agent.closedSet.end())
				continue; // Ignore the neighbor which is already evaluated

			if (map.isObstacle(neighbor)) { // if neighbor is obstacles, add to
closedSet.
				agent.closedSet.insert({ neighbor, MAX }); // fscore is set to
be max.
				continue;
			}
```

```
                        // The distance from start to a neighbor, diagonal 14, vertical or
horizontal 10.
                        uint dist_neighbor = abs((int)(neighbor.x - current.x)) + \
                               abs((int)(neighbor.y - current.y)) == 2 ? 14 : 10;

                        uint tentative_gScore = agent.gScore[current] + dist_neighbor;

                        if (tentative_gScore >= agent.gScore[neighbor])
                               continue;

                        agent.cameFrom[neighbor] = current;
                        agent.gScore[neighbor] = tentative_gScore;
                        agent.fScore[neighbor] = tentative_gScore +
manhattan_heuristic(neighbor, goal);
                        neighbor.fScore = tentative_gScore + manhattan_heuristic(neighbor,
goal);
                        /*
                        cout << "[" << neighbor.x << ", " << neighbor.y <<"]" <<" is
cameFrom " \
                        << "[" << current.x << ", " << current.y << "] ";
                        cout << "gScore: " << agent.gScore[neighbor] << "  fScore: " <<
agent.fScore[neighbor] << endl << endl;
                        */
                        agent.openSet.push(neighbor);
                }
        }
        return false;
}
```

```
/*****************************************************************************
reverse resumable A* (Backwards Search ignoring other agents)
the g value (measured distance to goal) is the true distance heuristic value.
if g value of requested node is not known, set goal at requestedNode.
*****************************************************************************/
bool get_true_distance_heuristic(Agent& agent, Map& map, const Node& requestNode)
{
	bool goal_found = false;

	//check requested Node
	if (map.isObstacle(requestNode))
	{
		cout << "<get_true_distance_heuristic>: Requested node is an obstacle.\n";
		agent.closedSet.insert({ requestNode, MAX });
		return false;
	}

	// start position is the goal position.
	Node start = agent.getGoal(); // reversed, goal is start.
	Node goal = requestNode;

	// Initially, only the start node is known.
	agent.openSet.push(start);
	// The cost of going from start to start is zero.
	agent.gScore.insert({ start, 0 });
	// For the first node, that value is completely heuristic.
	agent.fScore.insert({ start, manhattan_heuristic(start, goal) });

	while (!agent.openSet.empty())
	{
		Node current = agent.openSet.top();
		agent.openSet.pop();
		if (current == goal)
		{
			goal_found = true;
		}
		agent.closedSet.insert({ current, agent.fScore[current] }); // change
closedSet to vector.

		vector<Node> Neighbors;
		map.getNeighbors(current, Neighbors);
		for (Node& neighbor : Neighbors) {
			// if neighbors not exist, creat new nodes
			if (agent.cameFrom.find(neighbor) == agent.cameFrom.end()) {
				agent.cameFrom.insert(make_pair(neighbor, current));
				agent.gScore.insert(make_pair(neighbor, MAX));
				agent.fScore.insert(make_pair(neighbor, MAX));
			}
			// Ignore the neighbor which is already evaluated
			if (agent.closedSet.find(neighbor) != agent.closedSet.end())
				continue;
			// if neighbor node is obstacle, pust to closedSet and continue.
			if (map.isObstacle(neighbor)) { // if neighbor is obstacles, add to
closedSet.
				agent.closedSet.insert({ neighbor, MAX }); // fscore is set to
be max.
				continue;
			}
```

```
                        // if count diagonal neighbor, uncomment this code.
                        // uint dist_neighbor = abs((int)(neighbor.x-current.x)) +
abs((int)(neighbor.y-current.y)) == 2 ? 14:10;
                        uint dist_neighbor = 10;
                        uint tentative_gScore = agent.gScore[current] + dist_neighbor;
                        if (tentative_gScore >= agent.gScore[neighbor])
                                continue;

                        agent.cameFrom[neighbor] = current;
                        agent.gScore[neighbor] = tentative_gScore;

                        // if goal is already found, manhattan_heuristic is inaccurate to
measure fScore
                        // to understand, because manhanttan heuristic ignore obstacles.
                        // if goal is found, we simply add 20, why? for example, if we know
node A's fScore
                        // and node B is A's neighbor, and if A is the only parent of B,
then the cost of
                        // going from A to B is 10, so B_fSCore = A_fSCore + 10;
                        // in order for B to reach goal, it must go back to A,
                        // so add 10 to B_fScore again, total is 20;
                        if (!goal_found)
                        {
                                agent.fScore[neighbor] = tentative_gScore +
manhattan_heuristic(neighbor, goal);
                                neighbor.fScore = tentative_gScore +
manhattan_heuristic(neighbor, goal);
                        }
                        else
                        {
                                uint newfScore = agent.fScore[current] + 20;
                                agent.fScore[neighbor] = newfScore;
                                neighbor.fScore = newfScore;
                        }
                        /*
                        cout << "[" << neighbor.x << ", " << neighbor.y <<"]" <<" is
cameFrom " \
                        << "[" << current.x << ", " << current.y << "] ";
                        cout << "gScore: " << agent.gScore[neighbor] << "  fScore: " <<
agent.fScore[neighbor] << endl << endl;
                        */
                        agent.openSet.push(neighbor);
                }
        }
        return true;
}
```

5.3 Coordination and communication Component Architecture

As shown in the logic architecture, the coordination, as well as communication component is implemented from step 6 to 12. Basically, all agents coordinate and communicate inside a big while loop as show below.

```
// big while loop
while (!all_agents_find_goal) {
	if (!had_collision) {
		// fill hmap_t0 , hmap_t1
		for (auto& agent : agent_list)
		{
			Node current_n = agent->get_front_node_from_path();
			if (current_n != agent->get_current_node())
				agent->set_current_node(current_n);

			hmap_t0[current_n].push_back(agent);

			Node next_n;
			if (agent->getNextNode(current_n, next_n))
				hmap_t1[next_n].push_back(agent);
			else
				hmap_t1[current_n].push_back(agent);
		}
	}
	if (had_collision) {
		// only update hmap_t1, no change for the current node.
		for (auto& agent : agent_list)
			hmap_t1[agent->get_front_node_from_path()].push_back(agent);
	}
	// STEP 1: GOAL CHECK
	cout << "--------------- Do all agents reach destination? ----------\n";
	for (auto i = 0; i < n_agents; ++i)
		if (agent_list[i]->get_current_node() == agent_list[i]->getGoal())
			num_agents_at_goal++;

	if (num_agents_at_goal == n_agents) {
		all_agents_find_goal = true;
		cout << "Success! All agents at goal!\n";
		return 0;
	}
	Else cout << "--------------- Nope -------------\n"
	// STEP 2 COLLISION CHECK
	cout << "--------------- Is there any collision at next step? ---------\n";
	vector<vector<Node>> collision_nodes_pairs;
	// if collision, hmap_t1 is wrong do not do any thing to
	if (check_collision_type2(hmap_t0, hmap_t1, collision_nodes_pairs)) {
		had_collision = true;
		cout << "--------------- fixing collision -------------\n";
		fix_agents(hmap_t0, hmap_t1, collision_nodes_pairs, test_map);
		cout << "--------------- collision fixed -------------\n";
	}
	else
	{
		cout << "--------- Good, no collision, keep moving! ----------\n";
		had_collision = false;
		super_hash_map.push_back(std::move(hmap_t0));
	}
	hmap_t0.clear(); hmap_t1.clear();
	for (auto& agent : agent_list){
		agent->pop_front_node_from_path();
		agent->print_path();
	}
}// While loop end
```

6 Implementation Issues & Challenges

We now list some of issues and challenges:

1. Collision detection, two types of collisions;
    a. One is one node has more than one agents. This is easily to spot because one node can only contain one single agent at one time.
    b. This collision is hard to be seen, we call it face-to-face collision, at any time, there will be no more than one agent occupying a single node, but if agent A and B cross with each other face-to-face, meaning that A is walking from node 1 to node 2, while B is walking from node 2 to node 1. This is a collision.
2. Now we have collisions detected, what are do next? Fix agents' original path. Some of challenges in terms of implementing algorithms and data structure.
    a. The conflicted agents will coordinate with each other so that they won't collide soon.
    b. The conflicted agents will not and shall not influence other good agents' original path.
    c. Under rare or extreme circumstance, we will consider replanning good agent's path to make way for the colliding agents. For example, a good agent is blocking the only path of conflicted agents.
3. how to set window size, i.e. the depth of search. If map is huge, and agent has limited capability to efficiently search in real-time.
4. In terms of decoupled approach, how to implement coordination mechanism, communication component efficiently?

7 Supporting Systems Requirements

There is no specific requirement, as long as the hardware is capable of running c++ program.

8 Conclusions & Future Extensions

We hope to achieve some good result in terms of correct path planning for multiple agents. The problem is far more complex than I original thought. Therefore, I will not consider implement in GPU for now.

The priority is to correctly plan each agent's path and implement the simplest and yet robust coordination mechanism.

The future is unlimited. Once the mechanism, the algorithm and data structure are in place, we will tackle the performance in terms of speed and cost.

9 References

[1] Botea, A., Müller, M., & Schaeffer, J. (2004). Near optimal hierarchical path-finding. *Journal of game development*, *1*(1), 7-28.

[2] Matthews, J. (2002). Basic A* pathfinding made simple. *AI Game Programming Wisdom*, 105-113.

[3] Standley, T., & Korf, R. (2011, July). Complete algorithms for cooperative pathfinding problems. In *IJCAI* (pp. 668-673).

[4] Silver, D. (2005). Cooperative Pathfinding. *AIIDE*, *1*, 117-122.

[5] Stout, B. (2000). The basics of A* for path planning. *Game Programming Gems*, *1*, 254-263.

[6] Russell, S., Norvig, P., & Intelligence, A. (1995). A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, *25*, 27.