ECE9607: Collaborative A* Pathfinding

System Specifications Report

Yan Ge

Department of Electrical & Computer Engineering
The University of Western Ontario

2017, July 26

The report is dedicated to my Mom.

I want the world to know how grateful I am to be your child.

Table of Content

1 Overview

The last report of collaborative A* pathfinding focuses on the system specification. The implementation details along with fully commented source code are to be presented. The structure of this report is as the following. First, system scope and its major constraints are to be discussed, Follow by a general overview of system specification. We show the detailed implementation in section 3. This project is mainly based on David Silver's paper "cooperative pathfinding". C++ source code was written from scratch, and is solely based on Windowed Hierarchical Cooperative A* proposed in that paper. The code is completely unprofessional because this project is done in 3 weeks and during that time the author had 2 other final exams. I apologize for presenting such a poor-written C++ software. I am grateful to Professor Hamada Ghenniwa for his guidance and support. Without him, I would have never started my journey of pathfinding. And I borrowed the word "collaborative" from him.

1.1 Statement of Scope

The major inputs are,

- A gird map defined by height and width.
- A gird map may or may not contain obstacles which can be random or specific.
- Agents, with their own start and goal. However, starts or goals cannot be overlapped.

Processing functionality is perform multi-agent pathfinding. If system fails at pathfinding, system outputs error message, indicating which agent failed. Otherwise, system shows the path of every good agent on the map.

1.2 Major Constraints

At the moment, the project is fully focusing on the algorithm, not at real-world implementation level,

However, I can think of some constraints in terms of real-world deployment.

1. Map constraints. Map can be,
    a. A simple rectangular. (Amazon's warehouse)
    b. High-way. (Google's self-driving car)
    c. The sky. (Amazon's aerial delivery vehicle)

2. Cell of map, the dimensionality of map. For example,
    a. In 2D, cell can be rectangle, triangle, hexagon, or user-defined.
    b. In 3D, a cube, a sphere, etc.

3. Number of agents. This is an essential element of constraints. Because:
    a. For each agent, it needs to know all other agents' path.
    b. The collaboration is depending on number of agents.
    c. The communication or computation complexity is in $O(N^2)$. If I have N agents, each agent sends path to (N-1) agents. $O = N(N-1) = N^2$

4. How to define each agent’s neighbor,
    a. In horizontal or vertical, only 4 neighbors, if consider diagonal, 8 neighbors,
    b. if consider hex, 6; if consider star of David Neighborhood, etc.
    c. if consider circular, one can define nodes on the circle as neighbors.

5. Number of obstacles, level of randomness.
    a. Of course, one of the main goals is to avoid obstacles, If I have no obstacles, then I must avoid agents. If I have obstacles, I must avoid obstacles and other agents. Agents are simply moving obstacles.
    b. In real-world application, is obstacles still. I don’t think so. Because map may change in real-time, so do obstacles.

6. Size of agents.
    a. It matters in real-word application in terms of collision avoidance.

7. Speed of agents. In real-world application, the acceleration speed may vary.
    a. It is naive to avoid other agents solely based on relative positions. We must consider how fast can others start or stop moving, definitely not reach max speed immediately.

8. Move of agents.
    a. In this C++ program, it is simple horizontal or vertical move. I can upgrade it so agents can move in diagonal. Agent can also stay still.
    b. Of course, agents can move in any direction or not moving.

9. Time constraint
    a. Do you want agents to wait one hour before they start to move, or within one millisecond?

10. Computational constraints,
    a. How much storage does each agent have?
    b. Computing power, use GPU or not?

11. Uncooperative agents.
    a. In multi-agent soccer environment, we have enemy team, whose goal is to stop you from getting goal.
    b. If robot battery runs out, or communication failed, what we do?

12. Human – Agents interaction.
    a. In amazon warehouse, is it safe for human operator to enter that warehouse, or is it dead zone?
    b. Some folks want self-driving car, others prefer the fun of driving, how do they cooperate?

13. Vision of agent, how far can agent see, can predict its move and others.
    a. This is related to computational constraints; how far can agents plan ahead?
    b. Use a window, the idea is from David Silver's WHCA*.

1.3 Terminology & Definitions

• 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 resulting 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)	A 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	The basic unit of a path of an agent, include x, y coordinates, fScore, gScore.
• Path	Agent's path from start to goal
• 'X'	Obstacle or boarder 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.

2 System Specifications

A brief system specification is discussed in this section. Full details is shown in section 3.

2.1 Functional Descriptions

The system has several basic functions, including:

1. Defining a map with arbitrary size, random or user-specific obstacles.
2. Defining agents, each with a name, a start, a goal.
3. Simple A-star pathfinding for single agent.
4. Collaborative Pathfinding for all agents.

Specifically, the 4th function contains sub-functionalities, and I will show in detail with code present.

2.2 System Architecture

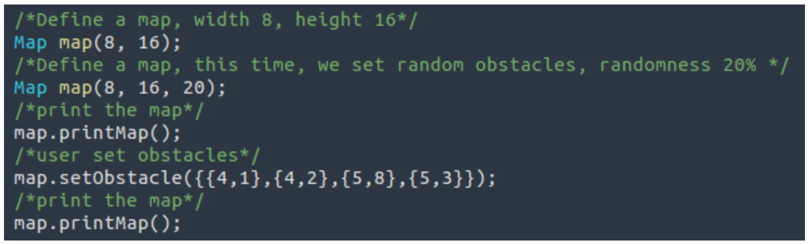
Overall objective is collaboratively pathfinding for all agents, make sure they don't collide, and reach goal safe and sound. System initiates when all agents are deployed at their own starting position on the same map. System shall end either when all agents reach goal position, or there is no pathfinding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

2.2.1 Deployment Architecture

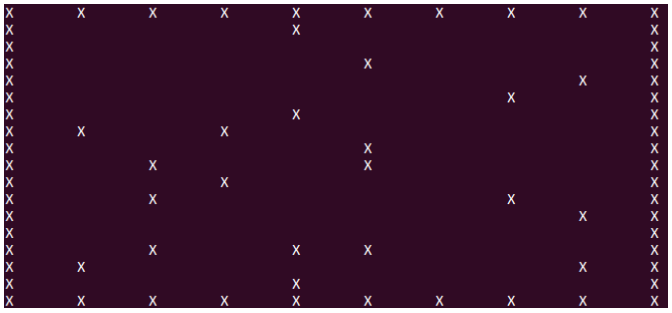
Agents are deployed on user-defined map. Now we discuss how the deployment works. The map size can be set arbitrarily. It may contain obstacles. System can set them randomly for you.

Let's define an empty map with width 8, height 16. Here, I made the map looks wider so you can see it clearly. Agents are deployed at each of their own starting position. Unfortunately, agents can only have one-letter name. Here, H means Professor Hamada Ghenniwa, Y is me. Obstacles can be set randomly or specifically.

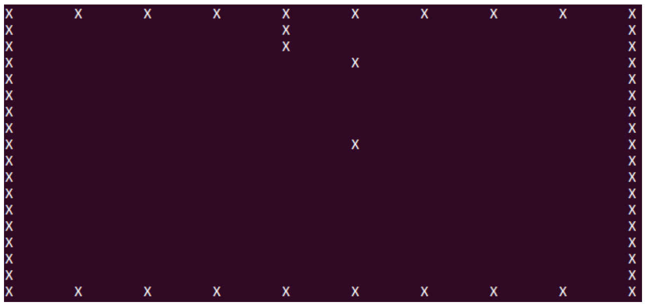
The following code creates two maps, left one has random obstacles, the randomness is 20%; the other one has user-defined obstacles at at {4,1},{4,2},{5,8}, {5,3}.



```
/*Define a map, width 8, height 16*/
Map map(8, 16);
/*Define a map, this time, we set random obstacles, randomness 20% */
Map map(8, 16, 20);
/*print the map*/
map.printMap();
/*user set obstacles*/
map.setObstacle({{4,1},{4,2},{5,8},{5,3}});
/*print the map*/
map.printMap();
```

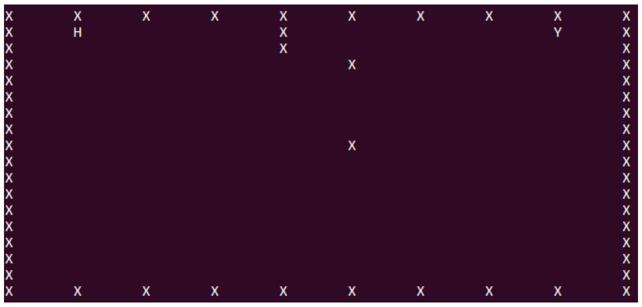


```
X   X   X   X   X   X   X   X   X   X
X               X                   X
X                                   X
X                   X               X
X                               X   X
X                           X       X
X               X                   X
X   X       X                       X
X                   X               X
X       X           X               X
X           X                       X
X       X                   X       X
X                               X   X
X                                   X
X       X       X   X               X
X   X                           X   X
X               X                   X
X   X   X   X   X   X   X   X   X   X
```

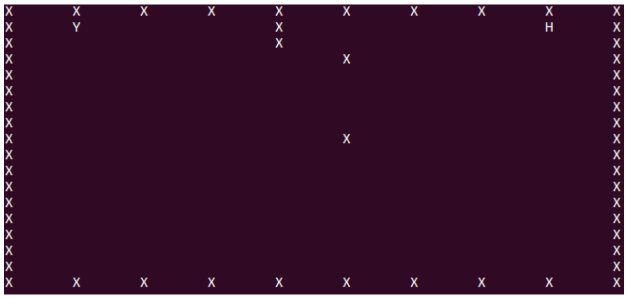


```
X   X   X   X   X   X   X   X   X   X
X               X                   X
X               X                   X
X                   X               X
X                                   X
X                                   X
X                                   X
X                                   X
X                   X               X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X   X   X   X   X   X   X   X   X   X
```

Now let’s create two agents, Professor Hamada Ghenniwa and his student, Yan Ge. H is Professor, Y is Yan. H starts at (1, 1), goal is (8, 1), while Y starts at (8, 1), goal is (1, 1). X stands for boarder or obstacles. Here we reuse the map we just defined with four specific obstacles. Below, on the left is the starting position, if pathfinding is successful, the goal is shown on the right.



```
X   X   X   X   X   X   X   X   X   X
X   H           X               Y   X
X               X                   X
X                   X               X
X                                   X
X                                   X
X                                   X
X                                   X
X                   X               X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X   X   X   X   X   X   X   X   X   X
```



```
X   X   X   X   X   X   X   X   X   X
X   Y           X               H   X
X               X                   X
X                   X               X
X                                   X
X                                   X
X                                   X
X                                   X
X                   X               X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X                                   X
X   X   X   X   X   X   X   X   X   X
```

2.2.2 Agent Architecture

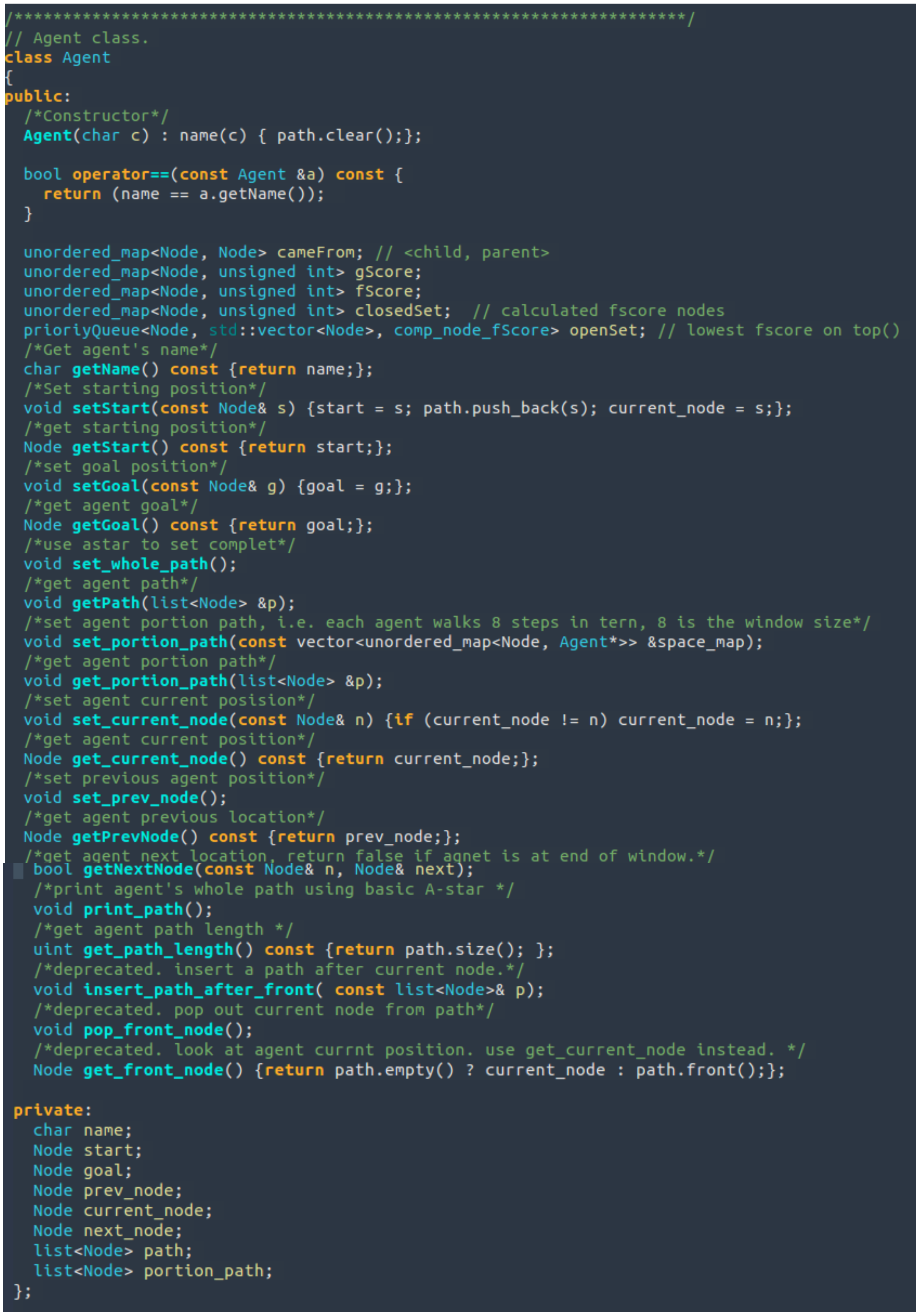
To seek high performance, Agent architecture is meant be designed as simple as possible.

On the contrary, the coordination mechanism in terms of solving interdependency issues shall be complex enough to consider all the scenarios. We want agents to 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. Collision 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. In future, I hope to 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, gScore hash map, fScore hash map, cameFrom hash map. As shown in the next page, the problem-solver for each agents is the public functions of Agent Class.

Most of the coordination, interaction are embedded inside helper functions which will be discussed in detail in the following section. A-star search, reverse resumable A-star, true distance calculation, collision detection and avoidance are separated from agent architecture. We will describe them each in detail in the coming section.

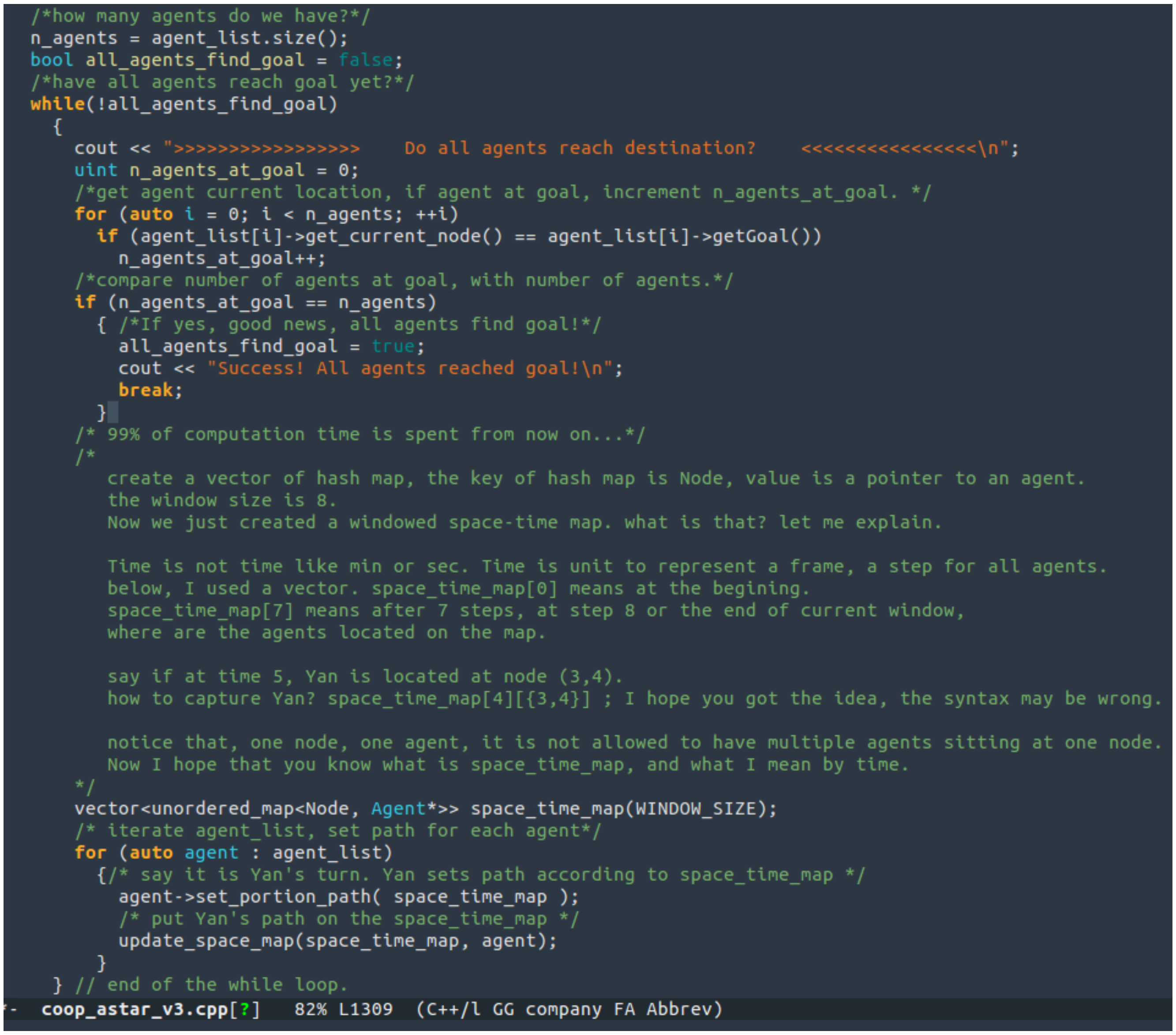
Again, code is completely unprofessional. It works. I hope you can forgive me for such poor-written C++ code.



```
/**********************************************************************/
// Agent class.
class Agent
{
public:
  /*Constructor*/
  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()
  /*Get agent's name*/
  char getName() const {return name;};
  /*Set starting position*/
  void setStart(const Node& s) {start = s; path.push_back(s); current_node = s;};
  /*get starting position*/
  Node getStart() const {return start;};
  /*set goal position*/
  void setGoal(const Node& g) {goal = g;};
  /*get agent goal*/
  Node getGoal() const {return goal;};
  /*use astar to set complet*/
  void set_whole_path();
  /*get agent path*/
  void getPath(list<Node> &p);
  /*set agent portion path, i.e. each agent walks 8 steps in tern, 8 is the window size*/
  void set_portion_path(const vector<unordered_map<Node, Agent*>> &space_map);
  /*get agent portion path*/
  void get_portion_path(list<Node> &p);
  /*set agent current posision*/
  void set_current_node(const Node& n) {if (current_node != n) current_node = n;};
  /*get agent current position*/
  Node get_current_node() const {return current_node;};
  /*set previous agent position*/
  void set_prev_node();
  /*get agent previous location*/
  Node getPrevNode() const {return prev_node;};
  /*get agent next location, return false if agnet is at end of window.*/
  bool getNextNode(const Node& n, Node& next);
  /*print agent's whole path using basic A-star */
  void print_path();
  /*get agent path length */
  uint get_path_length() const {return path.size(); };
  /*deprecated. insert a path after current node.*/
  void insert_path_after_front( const list<Node>& p);
  /*deprecated. pop out current node from path*/
  void pop_front_node();
  /*deprecated. look at agent currnt position. use get_current_node instead. */
  Node get_front_node() {return path.empty() ? current_node : path.front();};

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

3 Detailed System Specifications

The while loop below is the most outside layer of entire program. For each iteration, first test if all agents reach goal. If not, do the pathfinding. Main problem solver is the function “set_portion_path”. In the following section, I will mainly discuss this function. The interaction is through the space_time_map which agents can see each other and plan path according to others' existing path. The space_time_map is updated after each agent set its path.



```
  /*how many agents do we have?*/
  n_agents = agent_list.size();
  bool all_agents_find_goal = false;
  /*have all agents reach goal yet?*/
  while(!all_agents_find_goal)
    {
      cout << ">>>>>>>>>>>>>>>>>    Do all agents reach destination?    <<<<<<<<<<<<<<<<\n";
      uint n_agents_at_goal = 0;
      /*get agent current location, if agent at goal, increment n_agents_at_goal. */
      for (auto i = 0; i < n_agents; ++i)
        if (agent_list[i]->get_current_node() == agent_list[i]->getGoal())
          n_agents_at_goal++;
      /*compare number of agents at goal, with number of agents.*/
      if (n_agents_at_goal == n_agents)
        { /*If yes, good news, all agents find goal!*/
          all_agents_find_goal = true;
          cout << "Success! All agents reached goal!\n";
          break;
        }
      /* 99% of computation time is spent from now on...*/
      /*
         create a vector of hash map, the key of hash map is Node, value is a pointer to an agent.
         the window size is 8.
         Now we just created a windowed space-time map. what is that? let me explain.

         Time is not time like min or sec. Time is unit to represent a frame, a step for all agents.
         below, I used a vector. space_time_map[0] means at the begining.
         space_time_map[7] means after 7 steps, at step 8 or the end of current window,
         where are the agents located on the map.

         say if at time 5, Yan is located at node (3,4).
         how to capture Yan? space_time_map[4][{3,4}] ; I hope you got the idea, the syntax may be wrong.

         notice that, one node, one agent, it is not allowed to have multiple agents sitting at one node.
         Now I hope that you know what is space_time_map, and what I mean by time.
      */
      vector<unordered_map<Node, Agent*>> space_time_map(WINDOW_SIZE);
      /* iterate agent_list, set path for each agent*/
      for (auto agent : agent_list)
        {/* say it is Yan's turn. Yan sets path according to space_time_map */
          agent->set_portion_path( space_time_map );
          /* put Yan's path on the space_time_map */
          update_space_map(space_time_map, agent);
        }
    } // end of the while loop.
-  coop_astar_v3.cpp[?]   82% L1309  (C++/l GG company FA Abbrev)
```

3.1 Problem-Solver Components Description

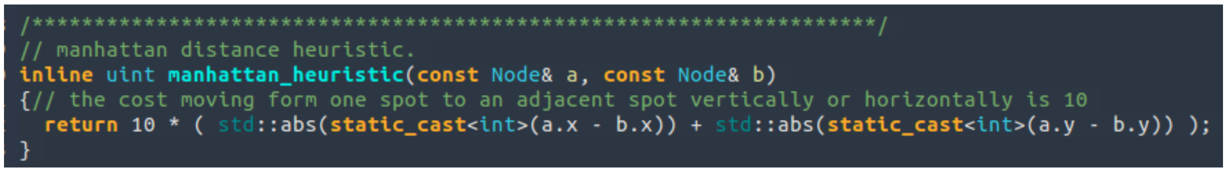
We have seen the “Agent” Class, it is meant to be simple. Imagine one thousand agents simultaneously path finding, if one single agent is capable of considering other 999 agents’ move, the system will crash sooner or later. Therefore, we let the interaction and communication components to handle complex interdependency issues, such as interest conflict, agent’s knowledge whether shared or not, agent’s capability.

The main duty of these problem-solvers which are embedded within each agent, is to provide individual agent with bounded capability and knowledge to help them interact and communicate with other, so that they solve problems collaboratively. In other words, it is the coordination mechanism to solve interdependency related issues, not single agent’s responsibility.

Since I am still a new-comer to CDS (Cooperative Distributed System), I make a lot of mistake especially when I am excited about this course. I will try not to be too excited.

3.1.1 Problem-Solver: Manhattan Distance Heuristic

Manhattan Distance method is to estimate the distance between two points. Point A is at (3, 2), point B is at (5, 7), then Manhattan gives absolute value of (5-3) plus absolute value of (7-2), the order does not matter here because of absolute value. Here, we multiply distance by 10.



```
/********************************************************************/
// 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)) );
}
```

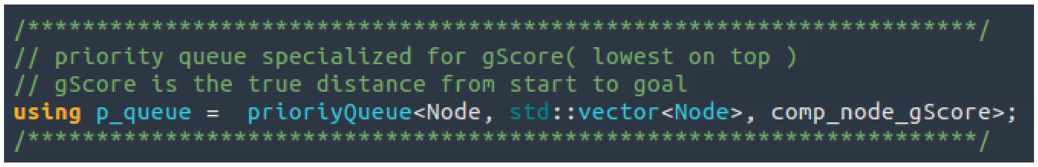
3.1.2 Problem-Solver: True Distance Heuristic

A natural question to ask first is what true distance is. You may find answer from David Silver’s paper “Cooperative Pathfinding”. In short, it is the distance taking account of obstacles. Manhattan distance simply ignores obstacles. Therefore I believe true distance is greater than or equal to Manhattan. When there is no obstacle, true distance is Manhattan distance.

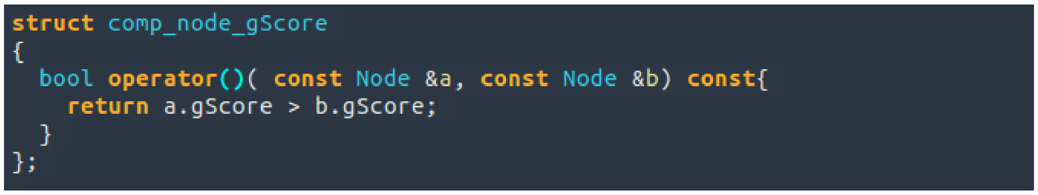
A second natural question to ask is how to calculate? Again, almost everything here is from David Silver’s well-known paper “Cooperative pathfinding”. In a word, reverse resumable A*. I will specifically discuss it in Section 3.1.1.4.

3.1.3 Problem-Solver: A* Search

I apologize for not giving detailed explanation on A-star. Because it is not the focus of this report. I was a new-comer to A-star one month ago. And it took me a while to understand and program it on my own. Wikipedia gives a good explanation and pseudo code. All the terminologies appeared in my code is based on that pseudo code. Only thing worth mention is that I implemented a customized priority_queue.

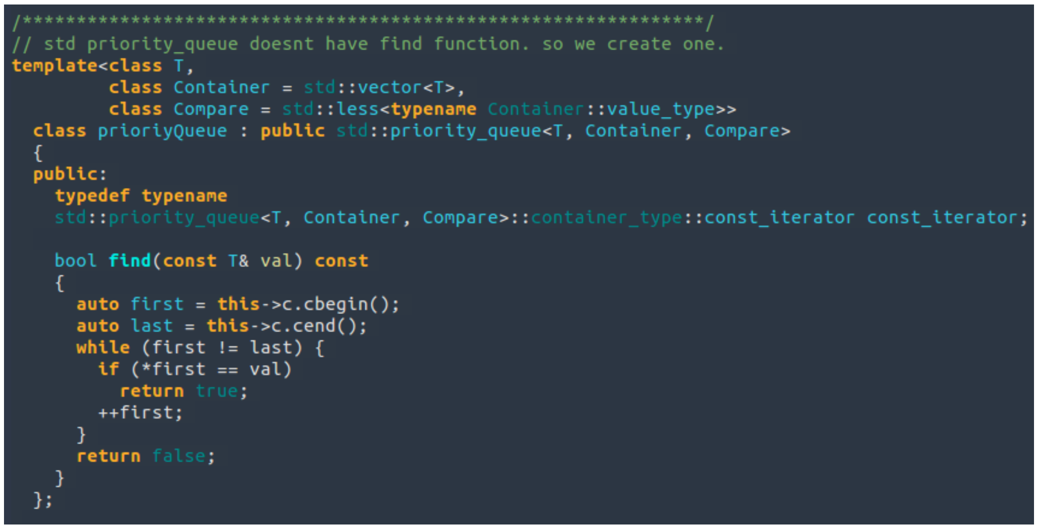


```
/*********************************************************************/
// priority queue specialized for gScore( lowest on top )
// gScore is the true distance from start to goal
using p_queue =  prioriyQueue<Node, std::vector<Node>, comp_node_gScore>;
/*********************************************************************/
```



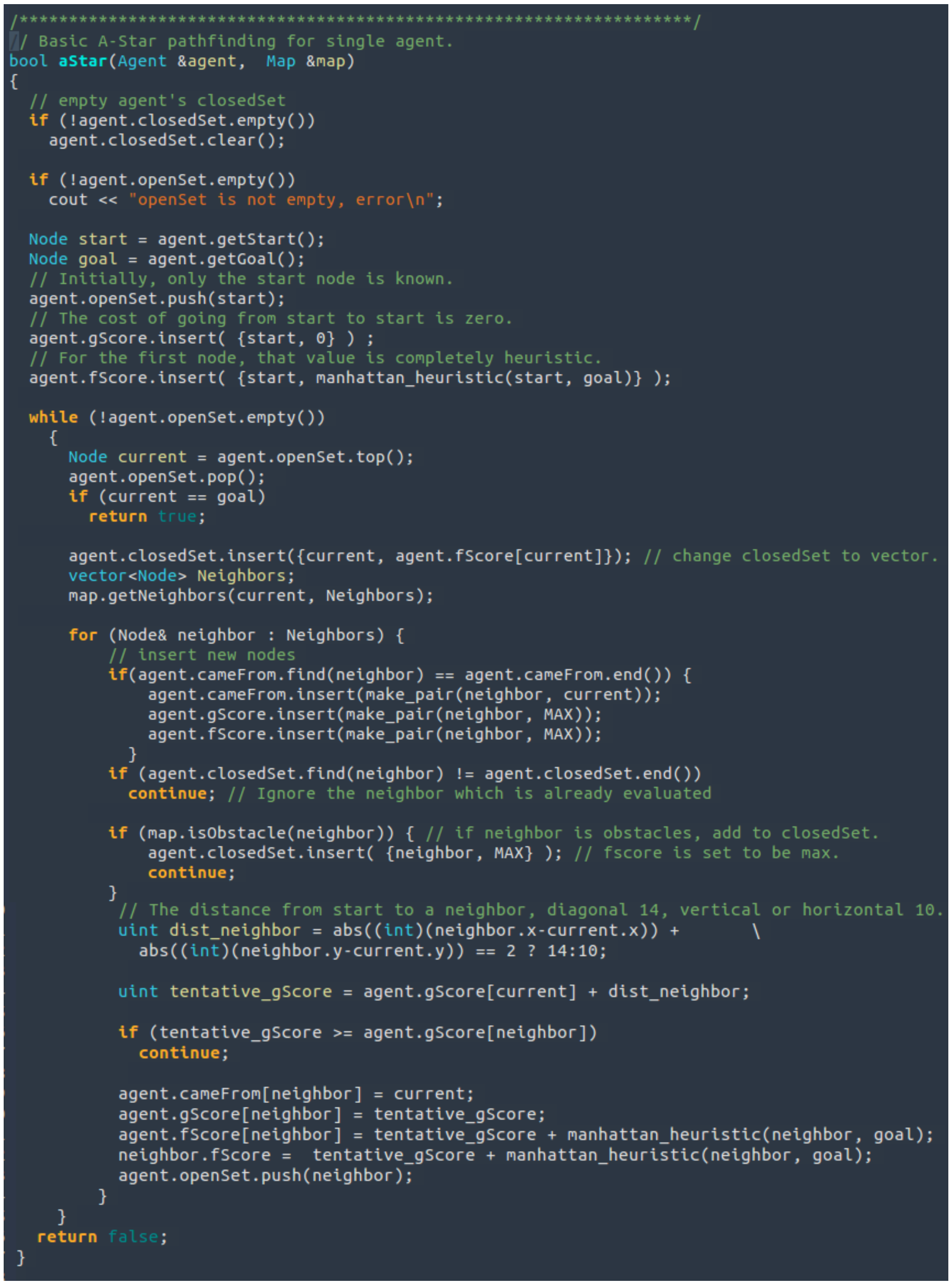
```
struct comp_node_gScore
{
  bool operator()( const Node &a, const Node &b) const{
    return a.gScore > b.gScore;
  }
};
```

Std::priority_queue is lacking a member function, find(); I think it is necessary for the purpose of this project.



```
/********************************************************************/
// std priority_queue doesnt have find function. so we create one.
template<class T,
         class Container = std::vector<T>,
         class Compare = std::less<typename Container::value_type>>
  class prioriyQueue : public std::priority_queue<T, Container, Compare>
  {
  public:
    typedef typename
    std::priority_queue<T, Container, Compare>::container_type::const_iterator const_iterator;

    bool find(const T& val) const
    {
      auto first = this->c.cbegin();
      auto last = this->c.cend();
      while (first != last) {
        if (*first == val)
          return true;
        ++first;
      }
      return false;
    }
  };
```

Next page shows my implementation of A* which is based on the pseudo-code from wikipedia. Here, I have considered diagonal neighbors.



```
/*********************************************************************/
// Basic 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);
           agent.openSet.push(neighbor);
         }
     }
   return false;
 }
```

3.1.4 Problem-Solver: Reverse Resumable A* Search

Back to the problem of calculating true distance heuristic, the solution is to use reverse resumable A-star proposed by David Silver.

The only difference I made is to do a full search once, no resume.

Meaning that before pathfinding, each individual agent has the full knowledge of the true distance score of all the nodes.

There is no resume in reverse A* search. Only search once, and never do it again.

Of course, it is exclusive and expensive if we have one thousand agents.

So, I plan to do it on GPU. And it it probably the only role of GPU. Is it realizable? You bet! The reverse A* search is independently calculated for each individual agent. Meaning agent do not interact or consider each other when performing reverse A* search.

Therefore it is perfect for GPU implementation.

The implementation is actually quite simple, you just reverse start and goal, meaning original start becomes goal, goal is start. And the gScore is the true distance heuristic.

So, when agent reached “goal” (original starting point). Do they stop? No, continue the search.

This is how to get the true distance for the rest of points? Simple, continue the search until the openSet is empty. Remember the A* search, it stops when agent find goal. Now it will do a full search until openSet is empty, i.e. no node to explore.

It is true that it is both stupid and exhaustive to calculate true distance score for every single node. Because agent might never ever go through most of the nodes calculated. In future, I plan to change that.

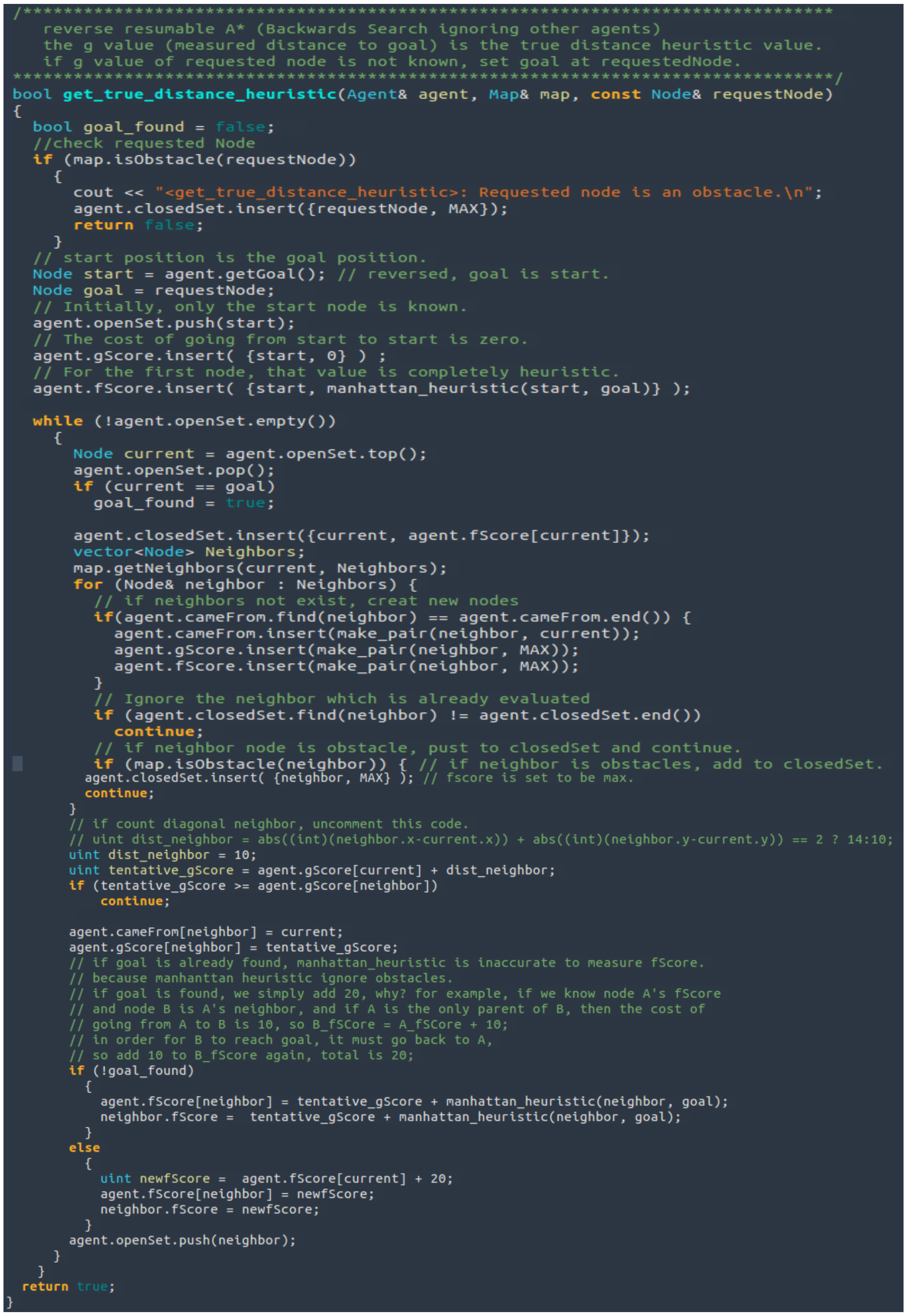
As I said, GPU will be responsible for all the calculation of true distance heuristic and I think it is fast in terms of overall performance.

Because if you want to resume A* search, the function call overhead might be a problem. Instead, just do one fully search, and forget about it.

The code is presented in the next page.

3.1.5 Problem-Solver: Collision Detection and Avoidance

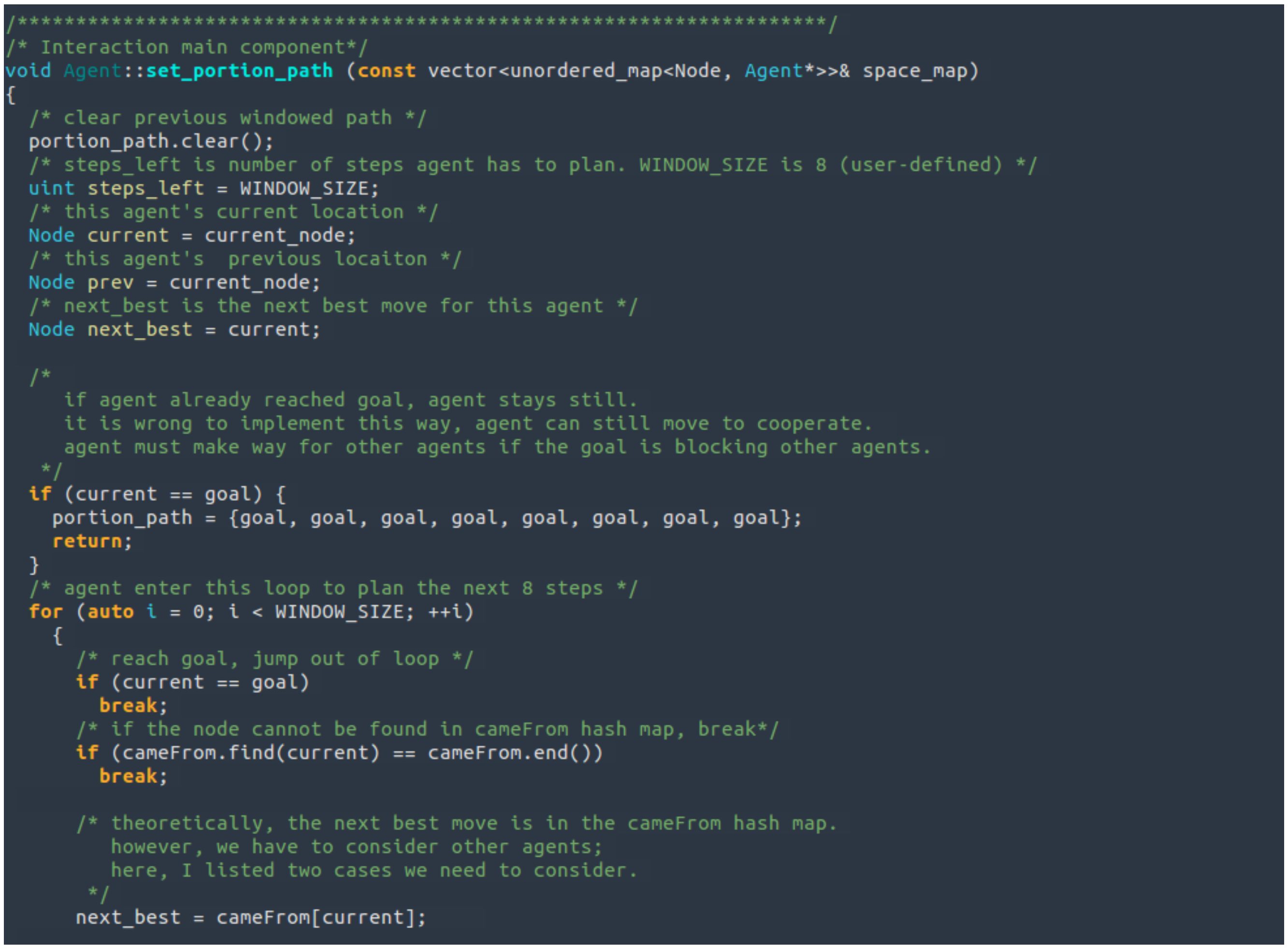
This problem-solver should be described in interaction component section. I mention it here because it is agent's responsibility to detect and avoid collision. However, it must through interaction and communication. Therefore, the detailed mechanism for this problem solver is illustrated in "3.2 Interaction component description".



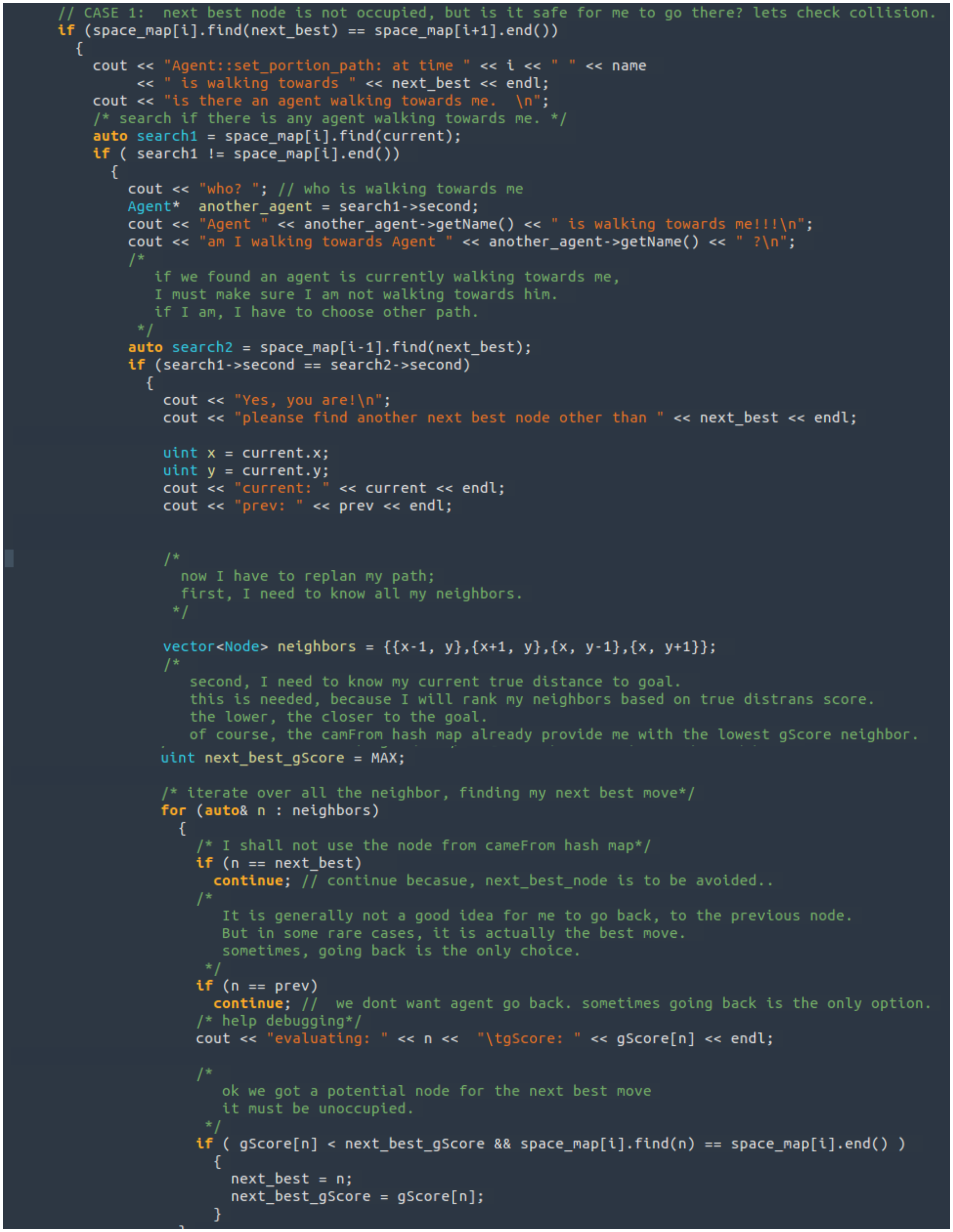
```
/*******************************************************************************
   reverse resumable A* (Backwards Search ignoring other agents)
   the g value (measured distance to goal) is the true distance heuristic value.
   if g value of requested node is not known, set goal at requestedNode.
*******************************************************************************/
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]});
      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.
      // 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;
        }
      agent.openSet.push(neighbor);
    }
  }
 return true;
}
```

3. 2 Interaction Component Description

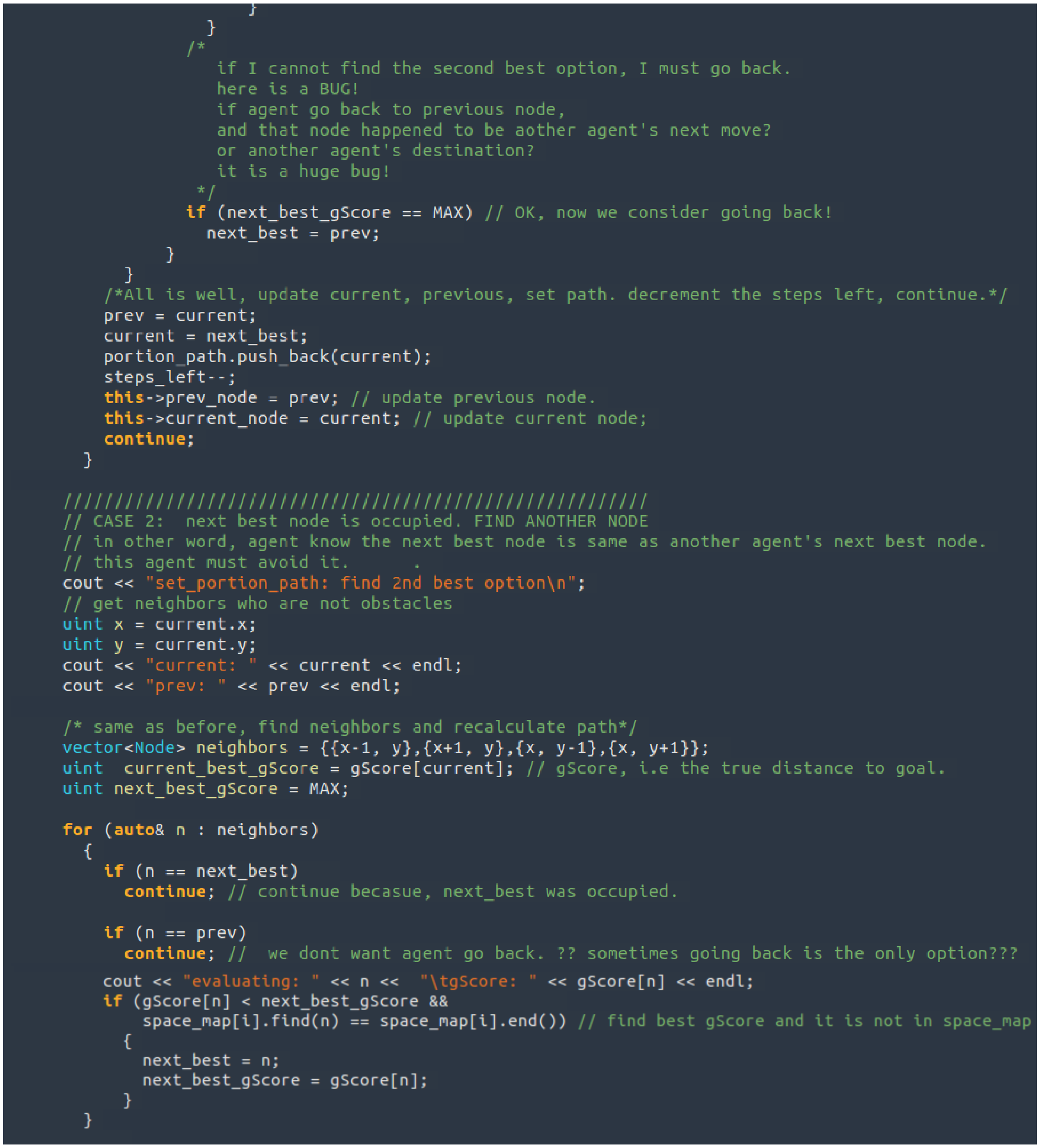
As I illustrated before, agents interact through “space_time_map”. Each agent plans windowed-time (8 steps) in turn based on other agents on that space_time_map. Then, each agent in turn leaves a trail on that space_time_map and waits for others to complete. When all agents finish planning their path in that window, the loop will then iterate, so on and so forth until all the agents reach goal. As shown in the beginning of section 3, agent in turn call this function set_portion_path(); and leave a trail on the space_time_map. The code presents itself, I hope the comment can do the talk.



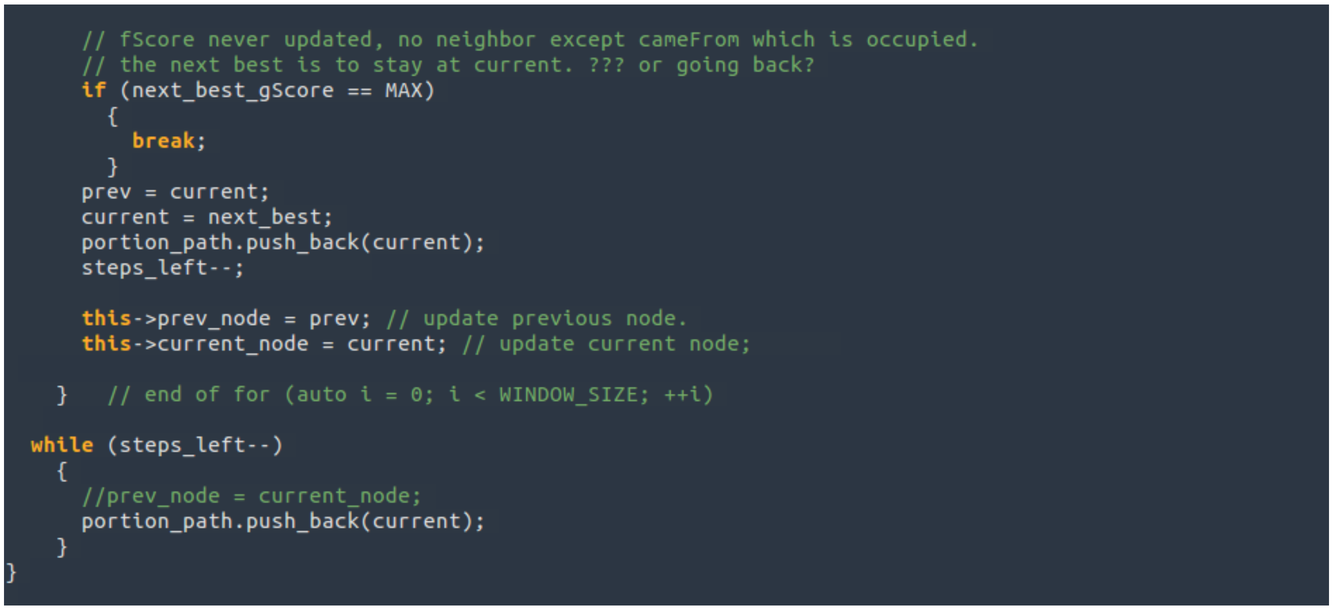
```
/**********************************************************************/
/* Interaction main component*/
void Agent::set_portion_path (const vector<unordered_map<Node, Agent*>>& space_map)
{
  /* clear previous windowed path */
  portion_path.clear();
  /* steps_left is number of steps agent has to plan. WINDOW_SIZE is 8 (user-defined) */
  uint steps_left = WINDOW_SIZE;
  /* this agent's current location */
  Node current = current_node;
  /* this agent's  previous locaiton */
  Node prev = current_node;
  /* next_best is the next best move for this agent */
  Node next_best = current;

  /*
     if agent already reached goal, agent stays still.
     it is wrong to implement this way, agent can still move to cooperate.
     agent must make way for other agents if the goal is blocking other agents.
   */
  if (current == goal) {
    portion_path = {goal, goal, goal, goal, goal, goal, goal, goal};
    return;
  }
  /* agent enter this loop to plan the next 8 steps */
  for (auto i = 0; i < WINDOW_SIZE; ++i)
    {
      /* reach goal, jump out of loop */
      if (current == goal)
        break;
      /* if the node cannot be found in cameFrom hash map, break*/
      if (cameFrom.find(current) == cameFrom.end())
        break;

      /* theoretically, the next best move is in the cameFrom hash map.
         however, we have to consider other agents;
         here, I listed two cases we need to consider.
       */
      next_best = cameFrom[current];
```



```
// CASE 1:  next best node is not occupied, but is it safe for me to go there? lets check collision.
if (space_map[i].find(next_best) == space_map[i+1].end())
  {
    cout << "Agent::set_portion_path: at time " << i << " " << name
         << " is walking towards " << next_best << endl;
    cout << "is there an agent walking towards me.  \n";
    /* search if there is any agent walking towards me. */
    auto search1 = space_map[i].find(current);
    if ( search1 != space_map[i].end())
      {
        cout << "who? "; // who is walking towards me
        Agent*  another_agent = search1->second;
        cout << "Agent " << another_agent->getName() << " is walking towards me!!!\n";
        cout << "am I walking towards Agent " << another_agent->getName() << " ?\n";
        /*
           if we found an agent is currently walking towards me,
           I must make sure I am not walking towards him.
           if I am, I have to choose other path.
         */
        auto search2 = space_map[i-1].find(next_best);
        if (search1->second == search2->second)
          {
            cout << "Yes, you are!\n";
            cout << "pleanse find another next best node other than " << next_best << endl;

            uint x = current.x;
            uint y = current.y;
            cout << "current: " << current << endl;
            cout << "prev: " << prev << endl;


            /*
              now I have to replan my path;
              first, I need to know all my neighbors.
             */

            vector<Node> neighbors = {{x-1, y},{x+1, y},{x, y-1},{x, y+1}};
            /*
               second, I need to know my current true distance to goal.
               this is needed, because I will rank my neighbors based on true distrans score.
               the lower, the closer to the goal.
               of course, the camFrom hash map already provide me with the lowest gScore neighbor.
            uint next_best_gScore = MAX;

            /* iterate over all the neighbor, finding my next best move*/
            for (auto& n : neighbors)
              {
                /* I shall not use the node from cameFrom hash map*/
                if (n == next_best)
                  continue; // continue becasue, next_best_node is to be avoided..
                /*
                   It is generally not a good idea for me to go back, to the previous node.
                   But in some rare cases, it is actually the best move.
                   sometimes, going back is the only choice.
                 */
                if (n == prev)
                  continue; //  we dont want agent go back. sometimes going back is the only option.
                /* help debugging*/
                cout << "evaluating: " << n <<  "\tgScore: " << gScore[n] << endl;

                /*
                   ok we got a potential node for the next best move
                   it must be unoccupied.
                 */
                if ( gScore[n] < next_best_gScore && space_map[i].find(n) == space_map[i].end() )
                  {
                    next_best = n;
                    next_best_gScore = gScore[n];
                  }
```



```
                  }
              }
            /*
               if I cannot find the second best option, I must go back.
               here is a BUG!
               if agent go back to previous node,
               and that node happened to be aother agent's next move?
               or another agent's destination?
               it is a huge bug!
             */
            if (next_best_gScore == MAX) // OK, now we consider going back!
              next_best = prev;
          }
      }
    /*All is well, update current, previous, set path. decrement the steps left, continue.*/
    prev = current;
    current = next_best;
    portion_path.push_back(current);
    steps_left--;
    this->prev_node = prev; // update previous node.
    this->current_node = current; // update current node;
    continue;
  }

/////////////////////////////////////////////////////////
// CASE 2:  next best node is occupied. FIND ANOTHER NODE
// in other word, agent know the next best node is same as another agent's next best node.
// this agent must avoid it.      .
cout << "set_portion_path: find 2nd best option\n";
// get neighbors who are not obstacles
uint x = current.x;
uint y = current.y;
cout << "current: " << current << endl;
cout << "prev: " << prev << endl;

/* same as before, find neighbors and recalculate path*/
vector<Node> neighbors = {{x-1, y},{x+1, y},{x, y-1},{x, y+1}};
uint  current_best_gScore = gScore[current]; // gScore, i.e the true distance to goal.
uint next_best_gScore = MAX;

for (auto& n : neighbors)
  {
    if (n == next_best)
      continue; // continue becasue, next_best was occupied.

    if (n == prev)
      continue; //  we dont want agent go back. ?? sometimes going back is the only option???

    cout << "evaluating: " << n <<  "\tgScore: " << gScore[n] << endl;
    if (gScore[n] < next_best_gScore &&
        space_map[i].find(n) == space_map[i].end()) // find best gScore and it is not in space_map
      {
        next_best = n;
        next_best_gScore = gScore[n];
      }
  }
```



```
      // fScore never updated, no neighbor except cameFrom which is occupied.
      // the next best is to stay at current. ??? or going back?
      if (next_best_gScore == MAX)
        {
          break;
        }
      prev = current;
      current = next_best;
      portion_path.push_back(current);
      steps_left--;

      this->prev_node = prev; // update previous node.
      this->current_node = current; // update current node;

    }   // end of for (auto i = 0; i < WINDOW_SIZE; ++i)

  while (steps_left--)
    {
      //prev_node = current_node;
      portion_path.push_back(current);
    }
}
```

4 Conclusion

Pathfinding is indeed very interesting, 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.

I would like to thank Professor Hamada Ghenniwa for guidance and support, this project will not exist without his teaching. And indeed I have learned a lot from Cooperative Distributed System. I apologize for presenting such a poor-written c++ source code. After this project, I would like to do some benchmark test and hopefully to implement multi-agent pathfinding on a GPU.

5 References

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