Planning and Navigation for a Game Bot

Giorgio Gori

February 19, 2016

Abstract

Games can force players into repetitive tasks in order to obtain in-game values such as currency or experience. Virtual worlds have different rules from reality but can be automated using using robotics techniques. This paper presents a method to improve the efficiency over the state of the art by real-time path-planning using the information provided by the environment.

1 Introduction

In-game values such as currency, items or experience are sought by players. Those can typically be obtained by killing computer-controlled agents in the game world. This process is repetitive, tedious, often fairly simple, and thus leads to investigating automation and robotics techniques where a piece of software is in control of the game character instead of the human player. This paper will refer to this software as the "robot".

In section 1 we provide an overview of the chosen game, the virtual world and the interface between game and robot. Section 2 contains the formal and informal description of the task to perform. In section 3 we will go over the current state of the art and related work in computational robotics. Section 4 contains a description of the proposed method. Section 5 contains experimental results and analysis. Finally, section 6 contains conclusions and future work.

1.1 Game Overview

We choose the game Guild Wars[1]. It is a relatively old game, released in 2005, that still has a modern interface and play style. Game areas are instanced, meaning that we can have the robot be alone with computer-controlled characters. This is good as the robot does not interfere with other players' experience and the robot only has to handle a relatively predictable world.

[1] `http://www.guildwars.com/`

1.2 Game World

The world space is two dimensional with continuous x and y coordinates, there is no vertical axis. Coordinates and distances are in unspecified "game units", which have no real-life counterpart. As a general idea, a character height is approximately 100 units.

There are two kinds of objects in the world, the first is obstacles. Those are represented as trapezoids like shown in Source Code 1. Those objects only serve as obstacles and other objects cannot move inside them. Their location is always fixed and known in advance.

The other objects are the agents, this includes the player character and any computer-controlled characters. As shown in Source Code 1, each agent has an unique ID, a position (x, y) in the world and many more attributes. Agents can move freely in the world with a constant speed, as long as they don't collide with obstacles or other agents.

The character controlled by the player or by the robot is an agent with the same characteristics as any other, but for clarity in this paper we will refer to "agents" as all moving objects that are not the character.

```
struct Agent {
    long id;
    float x;
    float y;
    // and more
}

struct MapTrapezoid {
    long id;
    float XTopLeft, XTopRight, YTop;
    float XBottomLeft, XBottomRight, YBottom;
}
```

Source Code 1: Game Structures

1.3 Sensors

The game offers most of its information through a 3D interface. Human players are good at interpreting it, but a robot would need a complex computer vision system to translate the images to data such as x,y coordinates.

Our approach instead is to read the raw data directly from the game memory. We find fixed memory addresses that hold pointers to game structures. For example we use an address that holds the pointer to the game's internal array of agents, which contains pointers to the Agent structures shows previously in Source Code 1. Declarations of those accessors are shown in Source Code 2.

```
Agent* GetPlayer();
vector<Agent*> GetAgents();
vector<MapTrapezoid*> GetMap();
```

Source Code 2: Declaration of accessors

Just like with real sensors, the robot and other agents have a limited vision range. The range limitation means that each agent only has knowledge of other agents within a fixed radius. This radius is known and is equal to approximately 1000 game units for agents and 5000 game units for the robot.

1.4 Actuators

Human players typically move the character by clicking on the ground in the game interface. The robot could send similar input by simulating a mouse click, but the location is dependent on the camera position, zoom, etc. Instead, we can trace the call: the game will translate the click coordinates from in screen to world space and then call an internal function. We locate this internal function and the robot can then use it directly as in Source Code 3.

```
void Move(float x, float y);
```

Source Code 3: Declaration of Move function

2 Problem Statement

We choose a location containing a specific kind of agents. In game terms, they are very powerful spell-casters, but in the game it is possible to equip a spell-invulnerability enchantment, which means that they can only use their weak basic attacks and few protection enchantments are enough to reduce the damage taken by the robot to zero. In other words, the agents are not able to kill the robot regardless of how many or them are attacking the robot. The robot still has to maintain its protective enchantments, but that is relatively easy and outside the scope of this paper.

Another important observation is that the robot has to kill agents in order to collect loot. The robot is equipped with area of effect (AoE) spells, meaning that it will do the same damage to nearby agent regardless of how many there are, as long as they are nearby. Using those spells takes time, so the most efficient way to collect loot is to gather as many agents as possible, and then kill all of them at once.

Finally, it is important to understand the behavior of the agents. They are all computer controlled, and they are fairly predictable. Agents spawn in random locations in the world in groups of 4. After they spawn, they will each move slowly and randomly. This behavior continues until they see an enemy,

and as soon as they see one, they will chase and attack it. In our case their enemy is the robot, and this is great for our purpose, as we want the agents to follow the robot. Note that their vision is shared between the group of 4, so as soon as one can see the robot, all members of the group will start chasing it.

We can conclude and describe the task informally as "Group all agents in the map", or "Move in such a way that all agents can see the robot at least once".

Formally, the problem can be described as follows:
Take as input a starting robot position (S_x, S_y), an end position (E_x, E_y) and a set $Agents = \{A_1, A_2, ..., A_n\}$ where each agent $A_i = \{id_i, x_i, y_i\}$.
Produce as output the shortest path $P = (p_1, p_2, ..., p_{m-1}, p_m)$ such that $p_1 = (S_x, S_y)$, $p_m = (E_x, E_y)$ and for at least one agent A_i in each group, $distance(A_i, p_j) < k$ for some $p_j \in P$.

Agents keep moving randomly and the robot can discover new agents as it travels, so it is not enough to compute the path once and use it. Instead, at each time step the robot should compute the shortest path P and move towards the next step p_2.

3 Related Work

3.1 State of the Art

The map is known in advance and never changing, and the agents typically appear in relatively similar positions, so the state of the art robots take the easiest approach: to specify in advance a list of coordinates that the robot should visit. This approach is referred as "Coordinate-based" or "Fixed-path". The advantages of this technique are its simplicity and robustness, but the obvious disadvantage is that the robot is blind to the environment. This kind of robot will not use agent positions, and because of that, will often perform unnecessary detours or miss groups of agents.

In this paper we will provide a solution to this blindness by using computational robotics techniques to make the robot react to the environment instead.

3.2 Related problems in Computational Robotics

The general problem immediately looks like the Traveling Salesman Problem (TSP). TSP has widely been explored in robotics literature, for example [3] gives an presentation of the problem and provides an heuristic to solve it.

The TSP also has some variants that are more relevant to the problem, [1] presents most of those variants and provides a solution to a similar problem that involves precisely visiting arbitrary shapes, the difference to this paper being that the problem is more difficult because of the arbitrary shapes, but the world is static, so computing a solution once is enough.

Two variants of TSP are particularly relevant. Generalized TSP (GTSP) involves visiting one node for every group as presented in [5] and this relates to our problem of only visiting one agent for every group. Close-Enough TSP (CETSP) consists in visiting each nodes by going to the neighborhood of the node, similar to visiting a circle centered at each node as shown in [6]. CETSP relates to our problem since we also only need to move within a set radius to the node. CETSP heuristic also often rely on a secondary problem, the touring of a sequence of circles as described in [2].

Overall, our problem is a combination of GTSP and CETSP.

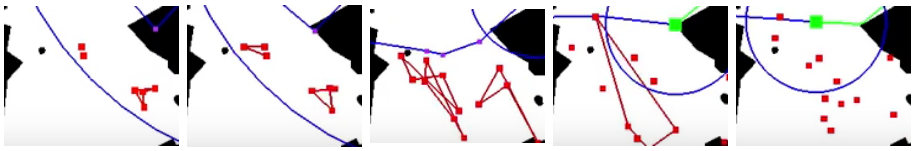
4 Proposed Method

The proposed method consists in three different components, first of all we cluster agents in groups of 4, then we compute shortest path between the groups using a Generalized Traveling Salesman Problem algorithm, and then we make an optimization pass.

4.1 Clustering

Groups of agents are fixed and set when they are created. We discovered that the agents of a group are created very close to each other, and start moving as soon as they are in vision of the robot. This allows the robot to perform a very simple and yet robust clustering.

At each time step the robot makes a list of all unclustered agents. If it can find 4 agents all within a fixed distance (experimentally set to 750 units) it will save them as a group. This process is performed in a greedy way until no clusters can be created. As soon as a group can see the robot it will start following, we don't need to consider it any more, so we remove it. This will save time later in the path planning. The life cycle of a group is shown in Figure 1.



(a) (b) (c) (d) (e)

Figure 1: Agents are represented by red squares and red lines connecting them represent groups. Two groups are created in (a) and (b), then they start moving randomly (c) along with a third group. Finally, the robot is seen by two of the three groups in (d) and they are removed. In (e) all groups saw the robot and were removed.

4.2 Generalized Traveling Salesman Problem

Once we have the groups, we need to compute in which order to visit them. We do that with a simple nearest neighbor approach. We construct a list of clusters, add the nearest to the path while removing it from the list, and repeat until the list is empty.

The main downside of the nearest neighbor algorithm is poor results towards the end of the tour, when most nodes are visited and hops become very long. However, we mostly care about the starting part of the tour, since that is where the robot will move. Towards the end of the path the data is also unstable, since we add groups as we see new agents. Nearest neighbor approach gives us stability where we need it the most at the begin of the tour, and it is a simple and fast method.

It may seem counterintuitive and unnecessary to compute the whole path with nearest neighbor and then only move towards the first neighbor, however computing the full path is important for the optimization pass below.

4.3 Path optimization

We perform two passes of path optimization.

First of all, we select the best agent to visit within every cluster. The nearest neighbor path generated in the last step favors agents close to the robot, but those are not necessarily the best. For every group in the path, we compute for every agent the distance from the previous group, to the agent, to the next group. The agent paired with the smallest sum of distances is selected.

The second step involves computing the final location to visit for each agent. Up until now, we used agents as nodes in the path, but the robot does not need to visit the agent, it only needs to be seen by them, or in other words, to be within a fixed distance to them. We use the 2-dimensional specialization of the 3-dimensional line intersection algorithm from [4]. There are two possible scenarios, shown in Figure 2, either the robot needs to make a detour to be seen by the Agent or it can just move straight. We use the above algorithm to detect the scenario and to find the point to visit. Finally we check if the selected point is valid in the navigation map, meaning it is not inside an obstacle, in case it is not valid, we revert this step and just use the agent position as destination.

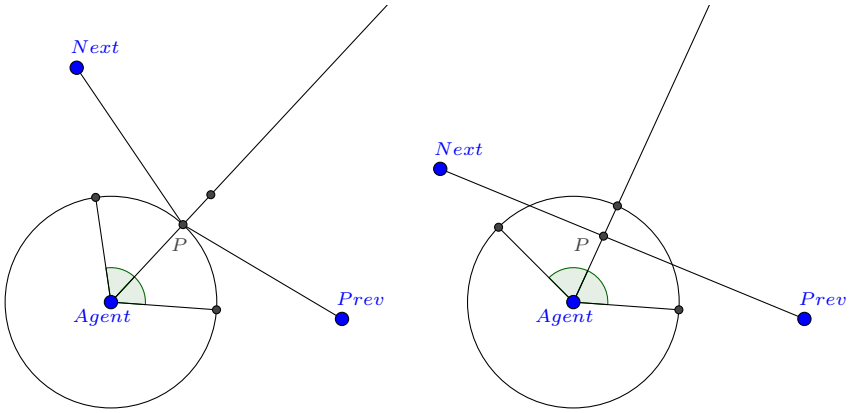


Figure 2: The two cases of position optimization, Left: The robot needs to make a detour to P to be seen by the Agent. Right: it can just go straight.

5 Results

One instance of the problem consists in loading a new map and performing the task of grouping the enemies. We record the time starting the timer when the map is loaded and stopping it when all agents are around the player character. We also considered the number of missed or lost agents. Finally, we noted failure, which can be caused by getting trapped by agents and being unable to move or by failing to periodically use the required enchantments and getting killed by the agents.

The player character is controlled by four different test subjects: an expert human with experience and knowledge of game mechanics, a beginner human with little knowledge of game mechanics and experience of the task, the state of the art coordinate-based bot, and our proposed approach. We perform 10 trials for each subject. We allow for some practice time to the human subject before the 10 test trials.

	Average Time (seconds)	Average Missed Agents	Fails
Novice Human	151	7.0	3
Expert Human	117	0.5	0
Coordinate-based	153	2.9	0
Proposed Approach	138	1.6	0

Table 1: Summary of the trial runs

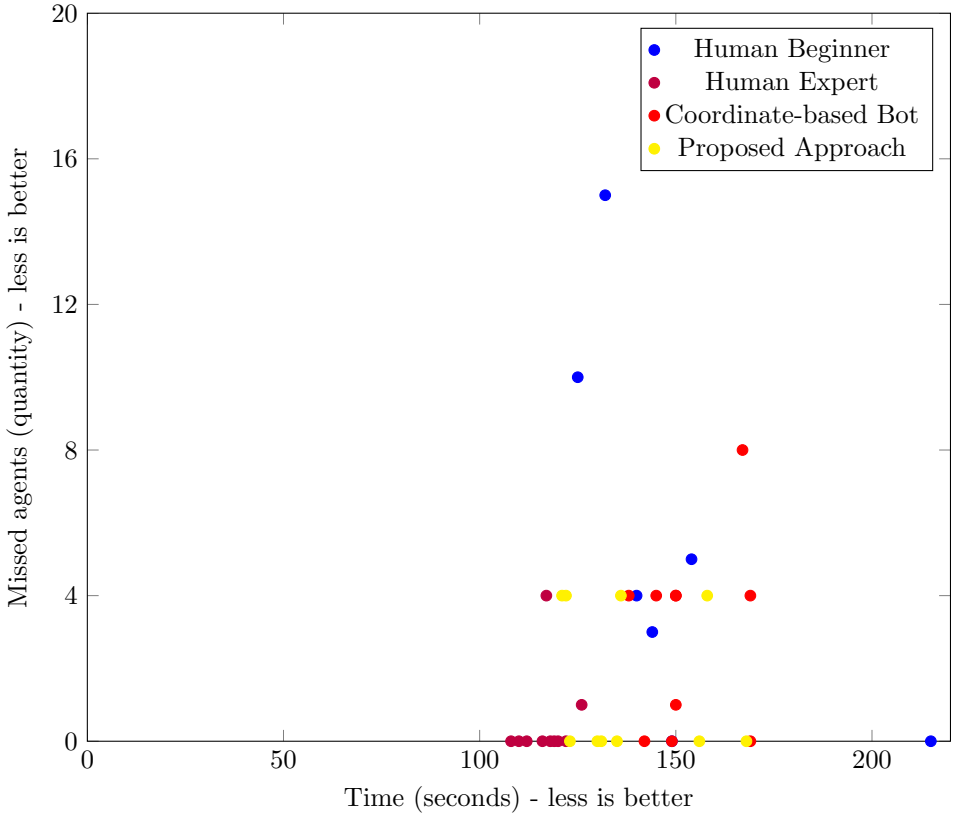


Figure 3: Plot of the different test trials. Different colors denote different test subjects as noted in the legend. The axis show time and amount of missed agents, in both axis less is better. The human beginner is missing three data points since those runs were fails.

5.1 Discussion

We show in Figure 3 the plot of each test trial. In this plot we are mostly looking for patterns of consistency and outliers. It is interesting to note how the proposed approach (yellow dots) yields generally good results, however there are three strong outliers with a much higher time than the others. The Coordinates-based bot performs consistently time-wise, but often misses a few agents. The "Human Beginner" is interesting in the fact that it is the most inconsistent: it can be relatively fast but miss a lot, or be really slow, and can still perform on par with the coordinate-based bot.

Table 1 is a summary of the runs and yet again gives a similar view on the performance of the different test subject. We can see how in average the Human Expert is the fastest, the Proposed Approach is about 20 seconds slower than that, and the Coordinate-based state of the art is an additional 15 seconds

slower. The Human Novice is similar to the Coordinate-based bot in terms of time, but misses a lot more agents in average and can also fail runs.

The main limitation of those results is their number, since we performed only 10 trials for each subject.

6 Conclusion and Future Work

As shown by our results, we implemented a path planning method which can outperform the state of the art. Our method can perform better than a human but can't reach the consistency and speed of a experienced player. The main issue is consistency time-wise usually caused by the robot getting stuck between agents. Future work will focus on obtaining more consistency, in order to do so the bot should not only perform the existing long-term path planning but also short term path planning to avoid getting stuck by agents and terrain.

7 References

[1] Sergey Alatartsev, Vera Mersheeva, Marcus Augustine, and Frank Ortmeier. On optimizing a sequence of robotic tasks. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 217–223. IEEE, 2013.

[2] Chang-Chien Chou. On the shortest path touring n circles. *International Journal of Advancements in Computing Technology*, 4(10), 2012.

[3] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[4] Ronald Goldman. Graphics gems. chapter Intersection of Two Lines in Three-space, pages 304–. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[5] Gilbert Laporte, Ardavan Asef-Vaziri, and Chelliah Sriskandarajah. Some applications of the generalized travelling salesman problem. *Journal of the Operational Research Society*, pages 1461–1467, 1996.

[6] William Kenneth Mennell. Heuristics for solving three routing problems: Close-enough traveling salesman problem, close-enough vehicle routing problem, sequence-dependent team orienteering problem. 2009.