Machine Translated by Google

AGS - searching the TileWorld space using an agent

Michal Glos (xglosm01)
Tom´aÿs Plach´y (xplach08)

December 19, 2021

1. Introduction

In this project, we solved the problem of searching the TileWord map using autonomous agents.
In total, we implemented 4 agents, described in more detail in section 2. With these agents, we then performed a series of experiments on different maps of different numbers, see section 5. Part In this project, there is also a graphic display of the map and agents in two images, where on the left is the entire map with obstacles and agents, and on the right is a part of the map that she was already searched by the agents. The component is also an environment for creating maps, described in more detail in section 4. This program also offers the possibility to save the search process as a video in mp4 format.

The functions implementing agent behavior have been optimized for speed, but some combinations of agent type, map search size, and number of agents are calculated very long, and large computing resources are needed to search the map. Python in minimum version 3.9 is required to run the program. In addition, ffmpeg is needed for saving videos.

2 Types of agents

In total, we implemented 4 different types of agents. These agents are described in this section, listed according to their abilities in ascending order.

2.1 Random

This agent moves on the field using random steps. With the help of arguments passed on the command line, it is only possible to modify the ability to move across the field and diagonally, as with all other agents. Since this agent does not show any sign of intelligence and cannot know if all available boxes have already been searched, its steps also include checking that the search is complete .

2.2 Naive

This agent already shows certain signs of intelligence, but his approach to searching is very naive.
The agent chooses the nearest undiscovered field (doesn't take into account obstacles bypassing the distance) as the goal he wants to reach. It then tries to apply all possible steps to its current position that do not lead to an obstacle or another agent and chooses the one that brings it the closest kc´ÿli. This agent has a weakness in that it cannot plan a path in such a way as to avoid possible obstacles, therefore its part is motion loop detection. If a cycle is detected, the agent will take a random step and with a 75% probability in each step will continue with the next random step. This agent can already search a random map in a relatively short time, however, if the agent encounters a position that requires several steps leading to the positions to achieve the goals , which moves away from the goal, the probability of bypassing the obstacle with each attempt decreases inversely proportional to the length of the path needed to bypass it. This agent also cannot determine if it has already searched all available boxes, therefore it is equipped with a function to check the completion of the search.

2.3 Smart

This agent works on the BFS principle of searching from an already discovered area. The function implementing BFS returns to the agent a list of the nearest undiscovered targets (the nearest field, on which it can discover another field with its input and). The agent already calculates the distance, including avoiding obstacles (when searching, the BFS algorithm does not take into account the positions of other agents). From this list, the agent then selects as a target the box on which it discovers the most undiscovered boxes with its entry. Then, using the A* algorithm, the agent finds the shortest path to the found goal, including bypassing the agent. If all boxes that the agent could find by climbing on the target are discovered, the agent finds another target. This agent can already find out by itself that it can no longer discover other boxes that are closed by obstacles, and it will stop.

Machine Translated by Google

2.4 Smart coop

This agent works on a very similar principle to smart, but the BFS algorithm does not only find the nearest field, to which the agent discovers a new field with its ascension, but also returns ÿ fields that have an agent's supervision of any fields not discovered. This creates a de facto outline of the discovered area in which the agent is located. The agent then chooses a target by asking all other agents about their target. The agent chooses its target so that the agent's field of vision, if any, does not intersect with another agent's field of vision in its target.
If no such field exists, the agent considers all possible fields obtained by the BFS algorithm.
From this set of possible targets, he then chooses the closest one with the most possible targets discovered on the field.

3 Launched

The script can be run using the command python3 agentsearch.py from the root directory and can be controlled using the following arguments passed to the script at startup:

- -c: Opens a map editor before starting the search itself, with the help of which obstacles and agents can be placed on the field in a custom arrangement . When running without this argument, the map is generated randomly (however, this can be influenced by other arguments).
  - letft-click By clicking the left mouse button on the map, you can change the occupied box - from an empty to the obstacle, from the obstacle to the agent and from the agent again to empty Poland.
  - "j": Press the "j" key to save the created map in the ./saved maps directory and run search and search.
  - "Enter": Press the Enter key to start the search on the created map, but the map will not be saved.
- -l [path/to/map]: Load a map from a JSON file at the specified address.
- ---diagonally: Allow the agent to move diagonally.
- --debug: Run slri'ut in debug mode.
- -m: Run the vmod script for measuring agent performance.
- -d: Number to z The number following this argument sets one (for a square map) to two map dimensions.
- -a: The number following this argument sets the number of agents (it has an effect only at random map generation).
- -o: The number following this argument sets the number of agents (it has an effect only at random map generation).
- -v: The number following this argument sets the range of the agent's field of view.
- -t: The chain after this argument sets the agent type (can be chosen from naive, random, smart and smart coop)
- -s: The string after this argument sets the path for saving the map.
- -r: The string after this argument sets the path for saving the search log. If we use this argument, the search video will be saved to the specified address.
- --dpi: The number after this argument sets the DPI of the recorded video scan.
- --animation-speed: The number after this argument will set the animation speed of the search.
- --frames: Set the number of frames of the saved video. set the number of frames with this argument of the saved video.
- -i: The string after this argument (from the set small, medium, large) selects the corresponding unsearchable map and starts a search over it.

Machine Translated by Google

3.1 Examples of execution

- python3 agentsearch.py -c To start the map editor (After pressing the Enter key, it starts search'av'an'ÿ).
- python3 agentsearch.py -l ./saved maps/experiment1 16 2.json -t smart coop To start and search the map of the first experiment with two smart coop agents.
- python3 agentsearch.py -d 10 20 -a 10 -o 50 To start a search of a randomly generated 10x20 rectangular map with ten 'naive' agents ' and fifty obstacles.

4 Creating maps

If you decide to run the script with the python3 agentsearch -c command, a map editor will open before the search starts. By left-clicking on any field on the graph, change its occupation from empty to obstacle. By clicking again, change it from an obstacle to an agent, and with another click from an agent again to an empty field. Press the "Enter" key to start searching this map.
By pressing the "j" key, the map will be saved in the ./saved maps folder and the subsequent search will be started.

5 Experiments

Naïve and random agents do not have deterministic behavior, therefore experiments with them will be run 5 times and the result will be the arithmetic average of all successful searches and. If at least one of the searches is unsuccessful, a value indicating the agent's success in percentage will appear behind the average in the table. Smart and smart coop agents have deterministic behavior, so experiments with them are only run once. The aim of the experiment is to find out how many steps the agents need to explore the entire map or to find out that the map cannot be explored in its entirety. The number of steps equal to three times the total number of squares on the map is considered to be a successful map search.

5.1 Experiment 1.

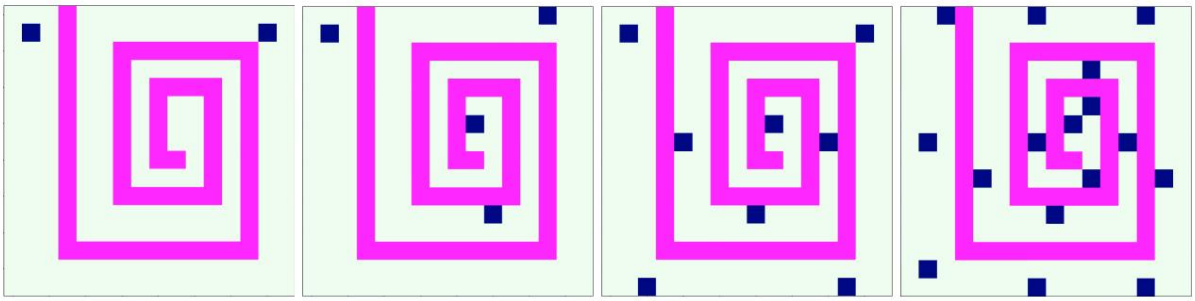

Figure 1: Maps used for the 1st experiment with different numbers of agents.

number of agents	random	naive	smart	smart coop (0%)
2	(0%) 114 44 52 (80%) 427.6	145.8	30	258.75
4				35
8				18
16		7	6	7

Machine Translated by Google

5.2 Experiment 2.

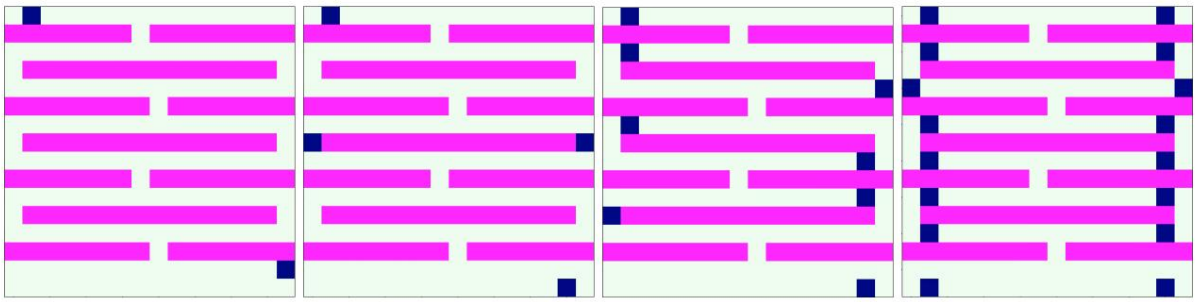

Figure 2: Maps used for the 2nd experiment with different numbers of agents.

number of agents	random	naive smart	smart coop (0%) (0%)
2 4	78,540 (60%) (0%) 28,447.67 (60%) 15.2 13 45.4 6		94
			35
			13
8 16		6	6

5.3 Experiment 3.

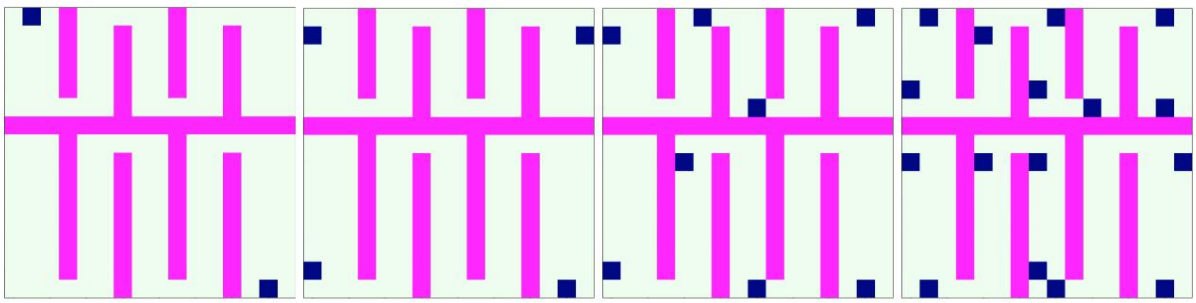

Figure 3: Maps used for the 3rd experiment with different numbers of agents.

number of agents	random naive smart	smart coop (0%) (0%)	51 501(60%)
2	(0%) 23 73 12 22.2 4		52
4			23
		18.8	11
8 16		4.6	4

Machine Translated by Google

5.4 Experiment 4.

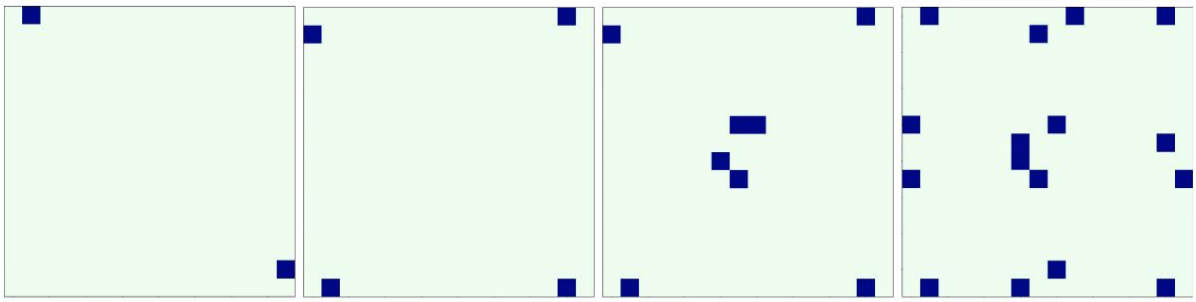

Figure 4: Maps used for the 4th experiment with different numbers of agents.

number of agents	random	naive smart smart coop 521.33 (60%)
2	53.8 45 255.8 26.4 95.6 11.4 22.2 4	38
4		19 19
		10 10
8 16		4 4

From experiments 1-4, it can be concluded that the behavior of the random agent does not affect the concept of the map too much, but it achieves the worst results in almost all cases. However, especially in experiments 2 and 3, he is able to search the field faster than the naive agent, and this is because the naive agent has a big problem in bypassing long obstacle. This problem can be eliminated by adding more agents so that there is almost no need to bypass obstacles, because behind each obstacle there is another It is an agent that explores the area. It can also be noted that the naive agent in number 16 achieved almost the same results as the intelligent agents, and this is because in this number the agents were able to search the entire map emÿeÿr only for using direct movements.

Intelligent agents (smart and smart coop) were more successful in every instance of the experiment, mainly because they can efficiently search any area that is not surrounded from all sides with walls.
It is interesting, however, that in the vast majority of cases, communicating agents did not defeat non-communicating agents.
The reason is probably the fact that for the maps that were part of the experiment, it pays to maximize the local search and not devote too much attention to the uniform search av´an´ÿ the whole map. In experiment 1, however, the cooperating agents in number 2 significantly outperformed the non-cooperating agents, precisely because the non-cooperating agent, located in the right in the upper corner he decided to search the same part of the map as the agent in the upper left corner. This resulted in the two agents meeting on the outside of the spiral and having to travel all the way to the center to search the rest of the map. However, the cooperating agent knew the position of the other's agent and therefore went in the other direction. This single case is so striking that although the non-cooperating agents were mostly able to search the map faster, the cooperating agents were able to complete all the experiments in fewer steps. Cooperating agents are generally immune to situations where several agents choose the same goal and therefore copy the same route.

Machine Translated by Google

5.5 Experiment 5.

To illustrate the behavior in which communicating agents outnumber non-communicating ones, we created a fifth map where (in small numbers of agents) more uncooperative Again, their agents unnecessarily copy the same route, whereas agents cooperatively divide the work and search the map more efficiently.

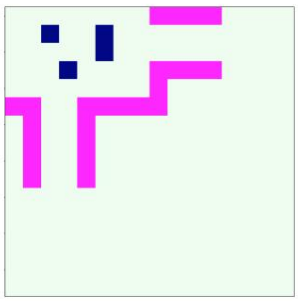

Figure 5: Map used for the 5th experiment with four agents.

number of smart smart coop agents 86 –		
2		59
4	45	31
	42	43
8 16	25	30

It follows that, although non-cooperative agents are often somewhat faster, in some cases (especially with a limited number of agents) their behavior it can lead to the choice of a very inefficient strategy, when the cooperating agents are superior in their consistency.

6 Close

The result of the work is several different agents, two of which show a certain degree of intelligence. When comparing them, however, it is not clear which of them achieves better performance. Probably, their performance could be optimized by a certain combination of their behavior. Another option for optimizing a cooperating agent would be to set a parameter how far the agent's goals must be from its possible goals in order to be above them he considered. The current implementation currently uses a distance of 2x the distance of the agent's supervision. Another possible improvement for the agent would be to plan longer trips, rather than only to Poland. However, ignorance of undiscovered shelves would be problematic and planning for discovered shelves would have to be somewhat stochastic. It would certainly be possible to apply neural networks to the agent's decision-making process, but their effectiveness here would be highly debatable.