MAPF and Ozobots

Final report

Seminar on Artificial Intelligence II

2018

David Nohejl

Chaman Shafiq

Věra Škopková

Problem we were solving

The original theme of our project was “Pick-up and Delivery using Multi-agent Path Finding (MAPF)”. It was planned to create a set of tasks in which robot, to whom would be the task assigned, should pick up some virtual object at a pick-up location and then he should deliver it to the delivery location as quickly as possible. After studying the problem and after the first attempts to implement MAPF algorithm on our own, we realized that the task is quite large and we decided to simplify it a bit.

Because the extension of simple MAPF to simple Pick-up and Delivery would not give us apparently more space to study anything interesting, we changed the theme only to MAPF and we started to focus on how the physical attributes of Ozobots can interrupt originally conflict free paths.

What we created

For this project we created three important programs. The first one is a JavaScript web application for designing the maps visually. The second program is the most important part of our work because it is the implementation of MAPF algorithm. As the basic algorithm we have chosen CBS (Sharon, Stern, Felner, & Sturtevant, 2015) and then we have extended it to k-robust MAPF (Atzmon, Felkner, Wagner, Barták, & Zou, 2017). And the last program we created is responsible for translating paths found by MAPF into instructions for Ozobots. All our codes are available on GitHub (GitHub - Věra, 2018), (GitHub - David, 2018) and (GitHub - Chaman, 2018).

Maps Designer

We have built a simple utility program to visually design the grid plan. It's a browser based JavaScript application that allows user to compose the grid by assembling various tiles. The final grid can then be exported as list of edges, which serves as input to the MAPF algorithm. It can also be directly printed. However, there are some unresolved difficulties when using the printed out map, namely the multipage printing of large map can be challenging, borderless printing can be an issue, and scaling is also an issue because Ozobots require pretty much one specific width of the line. Because of all these issues, hand painted maps still works the best for us. However, this tool was still very useful when designing the maps, and as an input to virtual test cases (just algorithmic runs), and to visualize the test cases.

MAPF Algorithms

We've implemented Conflict-based search (CBS) (Sharon, Stern, Felner, & Sturtevant, 2015), which is complete, optimal MAPF algorithm, in C#. There is an available implementation of CBS (GitHub - MAPF (doratzmon), 2018), actually ICBS, but it uses different representations of the grid map, and it takes some work to convert between the representations. We also wanted just the basic CBS, without many optimizations, because our experiments are relatively small (typically 8x4 grid with 2-4 agents), and we wanted to be able to easily modify the algorithm for our experiments. Additionally, we've implemented k-robust extension of CBS (Atzmon, Felkner, Wagner, Barták, & Zou, 2017). As a low level search we use A* with Manhattan distance heuristics. Initially we considered to implement or to use other algorithms like FAR, etc., but due to time constraints we did not.

Translation to Ozocode

Ozocode Generation

After finding plans for all agents, we needed to translate them into commands for Ozobots. Possibilities how to program Ozobots are very limited. It is because

Ozobots use the Ozoblockly language (Ozoblockly, 2018) that allows us to program by connecting colored blocks that represent single commands similarly like in children programming language Scratch (Scratch, 2018). Such style of programming can be fun for beginners, but it is quite uncomfortable for us. Firstly, building code from individual blocks makes programming slower and it is also a bit chaotic when writing more complicated code. There is not possible to write comments or to change the style of indentation. Secondly, more serious problem is that we need to translate many plans in very similar form into Ozocode and we are not able to do it automatically in Ozoblockly. Because of that we were searching for Java or Python interface to Ozobots but we had not succeeded, so we decided to implement our own program for the more comfortable generation of Ozocode.

When storing Ozoblockly program to a computer it has the form of text file with suffix “Ozocode“, and it consists of XML tags that represent individual blocks of the code. Interesting is that the blocks are not laid out one after another, but they are laid out recursively - always the preceding block contains the whole succeeding block in itself. The tags for individual blocks are mostly composed, all of them have the main tag that represents the type of the block and then it can have several subtags for setting the parameters. Because this structure is quite logical and regular, it was not difficult to implement functions that are able to generate an XML representation of the blocks and to compose the whole code for Ozobots from these functions.

Our program was written in C# and it is simple Console Application. For generation of the code for one Ozobot it requires an input file with a sequence of instructions which means the directions Ozobot should pick on the junctions (one after another). Program supposes Ozobot is moving on the grid so it works only with directions left, right, straight, backward and wait (no move). Every direction has a numbered constant which represents it and the input file has to contain only these constants – one constant per line.

The program reads the input file and it generates such Ozocode that the Ozobot (when put on the grid) goes exactly the directions described in the input file. For debugging purposes the program is able to include such commands into generated code that the Ozobot is able to say the name of the direction that he is actually picking and it is possible to include some light effects into Ozocode too.

Until we got access to the 5th section of the Ozoblockly editor we had to generate Ozocode as described above, it means by writing individual “move” and “pick direction” commands for every turn. The 5th section allowed us to use arrays and it made the whole generation of the Ozocode much simpler. Using the arrays, it is possible to create much shorter code consisted of a loop that goes through all fields of that array having different branches for individual directions. And such simple code was only the straightforward step to include other logic into the Ozocode. It was quite easy to add commands for sensors to detect if there is an obstacle in front of the Ozobot or to include commands for simple communication. And the best advantage of all of that is that codes for individual Ozobots differ only in the content of the array with directions. So it is not necessary to generate all commands in the generator separately, it is possible to design the code in the Ozoblockly editor and then to generate only the correct initialization of the fields of the array.

Obstacle Detection

In order to avoid collisions if something bad happens when executing the plans, we included simple obstacle detection in the code. Before the Ozobot starts moving to the next junction, it tests whether there is no obstacle. If there is an obstacle (not agent), Ozobot turns back and makes a step to the node he came from. Then he waits a random amount of time and then he tries to continue in executing the plan from the previous node. If there appears an obstacle on the last visited node, Ozobot will go back again until the first node of his path. When backtracking, if the node where Ozobot wants to go is already occupied by another Ozobot or by an obstacle then Ozobot waits.

Communication

Communication between agents is little bit complicated and tricky. The agent tries to communicate when it detects that there is something ahead of it by using its sensors. Hence let us describe it briefly how we made agents capable of communication with each other. We used obstacle detection first, so that we make sure that there is somebody with whom agent can communicate. Meanwhile, there was also another factor in our mind that what if there is an obstacle instead of a real agent? Because the agent can just sense and tell us that there is something in front of it without providing more information about it whether it is an obstacle or a real agent. So, we also have to handle this case during the attempt of communication with another agent. In short, after performing a lot of experiments, we figured out one way to differentiate between the real agent and the obstacle (if someone is interested, he/she can look the code). In case, if there is a real agent then communication takes place, but if there is an obstacle then agent steps back and waits there for a random amount of time then again tries to follow the path. Why it was tricky? Because when we tried to use communication feature in our main collision free path following program, then we realized that it is not easy to merge them altogether as we were expecting. We made a lot of attempts to merge them, but at last, we realized that it is not possible due to the limitation of storage, language and Ozobot hardware capabilities. There were some limitations which were impossible to solve according to our requirements. In simple cases, agents communicate with each other and take decisions according to the communication. There is also a FAQ about communication uploaded on GitHub (GitHub - Chaman, 2018).

Experiments and results

How to perform experiments with our tools

The way how to perform an experiment using our tools may look a bit complicated, so let us describe it in short. At first it is needed to use our Maps Designer and to draw the map. Then the application enables to export something that looks nearly like the input for our MAPF algorithm. But it cannot be used immediately, it is needed to replace all dashes (-) with spaces, add a line with the letter X and after that add information about the start and goal locations of all agents. For every agent there should be one row containing two numbers - number of the start and of the goal node. We number the nodes naturally, from left top corner to the right bottom corner, row after row, beginning with number 0.

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29

Numbering system of the grid

Now we have valid input for the MAPF algorithm which we can run. The program generates numbered text files, each of them represents a path for one Ozobot. But it is not usable yet, it is needed to get all these files to Ozocode Generator which will create the corresponding instructions for Ozobots.

As the last step it is needed to load every output of the Ozocode Generator into the Ozoblockly editor (on the web) and via it to store it into the account of the Evo mobile app. And finally, after turning the Ozobot on and connecting it via Bluetooth with the app, it is possible to run the program. The conditions for successful execution are that all Ozobots should always start the execution at the same time, they should be facing right and they should stay a bit behind their starting node.

We made many experiments and we filmed some of them. They are available on Google Drive (Video repository, 2018).

Big map – 4 Ozobots

This was probably our biggest plan. The map was a little bit bigger than A4 page and there were 4 Ozobots. Each of the Ozobots was starting from different part of the map and his goal position was on the other side of the map. Approximately in the middle of their way there was a bridge so the Ozobots had to order themself in order to go through that bridge. Although the plans for individual Ozobots were conflict-free in theory and although there were some attempts to insert some delays when Ozobots are not turning, there were still some collisions. We managed to set parameters of delays in such way that all of Ozobots executed their plans without collisions but it was very exhausting, we spend about 3 hours with this example. Successful execution can bee seen in a video.

Train with 4 Ozobots

In this experiment we divided 4 Ozobots into 2 couples with similar start and goal positions. Then the plans for Ozobots in the couple were very similar and Ozobots were creating two “trains” during execution (going always the same way, just one a node behind the second). For a while they were connected into one longer “train” also. There was possible to observe how they were asynchronous a bit but they still managed to execute the plans successfully. This example is also available to see in the video.

ZigZag

This example shows how serious delay can be caused by turns. Two Ozobots start synchronously and both of them should move to the next node 5 times. The path for the first Ozobot has 5 nodes in one row while the second Ozobot has to make turn on every junction. At the end of the execution the first Ozobot finishes one step before the second Ozobot. Video to this example is also available.

K-robust example

In this example, Ozobots have it really simple! There is single path from start to finish, no crossings, nothing. Just a few turns. But the catch is that the robots’ starting points are next to each other, and so are their goals. So in optimal solution, they will create a train and closely follow one another. As we found out, in real world, they sometimes crash when one turns slower than the other, and sometimes our built-in collision detections kicks in and one robot goes step back before resuming his path. To resolve this issue, we deployed the 1-robust solution (which on this map really means that the second robot is delayed by one step). And then everything works fine and Ozobots successfully reach their goals. All these situations are visible in our video repository.

Step Back

As described in the section about what we did, we tried to execute some plans with obstacles. In practice there is the problem of sensitivity of the proximity sensors so making step back when detecting another Ozobot did not work properly in all cases. But it worked good, when using a bigger obstacle, actually we are able to demonstrate this example live using one Ozobot and a hand. Putting hand always on the node Ozobot is actually going to visit, we are able (in extreme case) to force him to return from the last but one node to the first one.

Communication

During the attempts of communication between ozobots, we confronted a lot of challenges. Few of them were little annoying while, on the other hand, some of them were impossible to achieve according to our expectation. One of them was; we were thinking that communication part is working fine separately and it will also work when we will merge it with our main MAPF program. But when we tried to merge it, then we saw that there are some communication code compatible issues are occurring and then, when we tried to resolve them other function started to misbehave. In a technical sense, Ozobot communicates with other when it is doing nothing or executing asynchronous instructions. It means Ozobot do just some things in combination parallel not all. For example, we were trying that during the execution of finding next intersection or line end if you detect obstacle or other agent then behave according to the situation. It means, if there is an obstacle then step back and if there is agent then communicate with him. When we study the documentation thoroughly, then we got to know that this thing is not possible and the agent can do this task just with the combination of fewer instructions. In simple words, it is possible to execute the line following instruction and the sensors to detect obstacle or agent for communication. In short, we extract out the communication part from our MAPF algorithm and let it separately. The annoying thing was to load the program through flash because of communication. There is also a way to load the program through Mobile app rapidly, but mobile app currently doesn’t support the communication feature. Therefore when we tried this possibility, the app destroyed our code and just showed those instructions which were supported by it. Hence, all the time, after it we used flash for loading the programs in Ozobots. The second problem with Ozobot was that it didn’t see the other agent coming towards him, but it efficiently detected the other kind of materials. Thus, we also faced this problem all the time due to the inefficiency of detecting other ozobot. Just to see the communication between them almost every time we used to start the Ozobot from start, so that we can see whether they communicating perfectly or not. In a nutshell, we achieved, which we wanted by using algorithms, but we couldn’t get it fully by communication as we were expecting because of the Ozobot limitations.

Problems

Quality of the line

The first “physical” problem we were facing when executing plans on Ozobots was the right detection of the line that Ozobot should follow. Ozobots detect correctly line only of a given width which is around 5 mm. When the line is visibly thinner or thicker it may cause problems. Moreover, the line should be homogeneous. If it is broadening, it can be detected as junction sometimes. Also the color of the line should be homogeneous, the change of the color of the line can be also interpreted as a junction.

Junction detection

During the whole execution of the plans on Ozobots we strongly rely on proper junction detection. Missing the junction would destroy our plans irrecoverably. It means every junction should be drawn properly - it should be really a cross with right angles.

We suppose we have a grid with some nodes missing. It means in our maps there can appear situation that there is a node in which only one direction can be chosen. But such junction is not detected as junction by Ozobots, Ozobots behave like it is a straight line. Because the detection of such situations in the program would be annoying and it has no theoretical or practical meaning, we decided to solve this problem by representing every junction like a cross, adding small pieces of the line at positions where the neighboring nodes are missing.

Imperfection of sensors

As mentioned above, we tried to improve our Ozocode with some obstacle detection using proximity sensors. Theoretically, it worked well. We found some limit how near the object should be if it should be detected and we designed the nodes of the grid with the appropriate distances between each other.

But in reality it didn´t work as easy. The Ozobots detect bigger obstacles like a human hand properly, but they have problems to detect each other. If they meet exactly face to face they have nearly no chance to see each other, probably because of the material on the surface of the Ozobots. Such problem made our code for detection of obstacles nearly useless.

Multiple turns on one junction

When implementing obstacle detection, there was needed to be able to do more than one turn on one junction. It is not possible with “Pick direction” Ozoblockly commands, they allow only one turn and then Ozobot does not stay on the junction any more. Because if it, it was needed to replace all these commands with low level commands “Rotate a given angle”. But after rotation Ozobot may lose control of the line he is standing on so it was needed to also replace all “Follow line” commands with “Move forward until the line is found” commands.

Loading code into Ozobots

Actually, there is no other possibility how to load Ozocode into Ozobot except for loading it via the Ozoblockly web editor. This editor allows two possibilities of loading. The first one is so called flashing. It means that Ozobot is put on the screen and the program is loaded through a sequence of color blocks. This way of loading takes a lot of time and it also needs enough access to the internet. It is nearly impossible to load program successfully when, for example, listening to music on YouTube.

The second possibility of loading Ozocode into Ozobot is via mobile applications. It is quicker and more comfortable and it is possible to have more programs in the app and to switch between them at any time. But it has also some limits. The mobile phone has to have higher versions of Android, it is not possible to run it on for example Android 4. Moreover, it is needed to use Bluetooth in the phone because the app communicates with Ozobots via it.

Starting the Ozobots

The next problem is to start the Ozobots at the same time. When using a mobile app, every Ozobot needs different phone. It is possible to connect more Ozobots to one mobile but then they have run the same program. When starting Ozobots by double-pressing of the power button then it is needed to have a skillful hand to manage it properly. Otherwise the demo program is started instead.

Connection to other algorithms

We also attempted to connect our maps to the foreign CBS implementation (GitHub - MAPF (doratzmon), 2018), but there was problem of different map representation. It was because we always explicitly name the couples of nodes which are connected by an edge while this algorithm expects edges between all present nodes automatically. We managed to convert our maps into this representation using auxiliary nodes between all couples of connected nodes but there were some plans we were not able to convert their outputs back to our maps. It was when there was any step in which an agent was waiting on the auxiliary node, then it was impossible to convert the plans back.

Future Possible Work

There are two main directions in which we can imagine future work. One is to turn our ad hoc utilities into a kind of framework or workbench where different MAPF algorithms and their modifications can be tested both on in silico and in the real world using the Ozobots. Second direction is to focus on improving the robustness of execution in Ozobots by either further investigating online resolving of unexpected collision, possibly with communication, or making more robust plans. We’ve briefly touched on that with k-robust CBS, but more experiments are needed.

To realize the “framework” idea, several steps need to be done. One is to finalize the designer application. Work out printing maps out of designer, allow custom grid sizes, specify the starting points, etc. Other is to include a simulator, or a way to visualize ideal execution of the found solution in computer. One of the most problematic parts of the experiments is synchronized start of the agents. There should be a way to easily start all agents at once. Existing application for Ozobots allows to connect to multiple robots, but it seems we can only execute the same program. Last but not least, currently our solution is several applications, which you have to run one by one to get to the final code for Ozobot. This should be integrated into one user friendly application.

Tasks Distribution

David Nohejl	**Chaman Shafiq**	**Věra Škopková**
Map designer CBS implementation kR-CBS implementation Study of theory	Communication Obstacle detection Step back in Ozocode Design some test cases Review final report	Ozocode Generator Obstacle detection Steps back in Ozocode Presentations Final report Design of experiments Interface design Organization of the team

References

1. Atzmon, D., Felkner, A., Wagner, G., Barták, R., & Zou, N.-F. (2017). K-Robust Multi-Agent Path Finding. *Proceedings of the Tenth International Symposium of Combinatorial Search*.
2. *GitHub – Chaman*. (2018). Retrieved from https://github.com/chamanshafiq
3. *GitHub - David*. (2018). Retrieved from https://github.com/DavidNohejl/bot-track-designer

4. *GitHub - MAPF (doratzmon)*. (2018, May 13). Retrieved from https://github.com/doratzmon/MAPF
5. *GitHub - Věra*. (2018). Retrieved from https://github.com/verka383
6. *Ozoblockly*. (2018). Retrieved from https://ozoblockly.com/editor?lang=en&robot=evo&mode=4
7. *Scratch*. (2018). Retrieved from https://scratch.mit.edu/
8. Sharon, G., Stern, R., Felner, A., & Sturtevant, N. (2015). Conflict-Based Search for Optimal Multi-Agent Pathfinding. *Artificial Intelligence, 219*, pp. 40-66.
9. *Video repository. (2018).* Retrieved from https://drive.google.com/drive/folders/18fG_vvyCQUzU4GKC1rxzIuQ2lJ4lqujp?usp=sharing