Seminar of Artificial Intelligence II (NAIL052)
Team 2: Felipe Vianna, Yuu Sakaguchi, Surya Prakash Chembrolu
Project Topic: Manufacturing – multi-agent path finding
15.5.2018

Final Report

1. PROBLEM DESCRIPTION

This task can be described as multiagent pathfinding for manufacturing, simulating a modern approach in which the products can find the operations scheduled for them. Using scheduling algorithms it's possible to define sequence of operations regarding constraints and minimization of conflicts. Later, it's possible to define collision-free paths trying to minimize total number of steps of all agents. The motivation of our proposed project is the possibility to apply such concepts in Production Planning and Control, combined with IoT tendency in modern industry.

2. IMPLEMENTATION

After preliminary tests and studies for better understanding of our problem and the Ozoblockly environment, we divided the project into three major parts: Scheduling, Path Finding and Ozobot code. Each member of the group became responsible for one part, and their implementations are described below.

1. Scheduling (Surya Prakash Chembrolu)

The goal of our micro project was to demonstrate the simulation of manufacturing processes using ozobots with the central theme of multi-agent path finding. According to our vision, the problem is seen as each agent (ozobot) has to go through certain set of actions or operations at different locations and those operations are at fixed nodes on the grid. Looking at the problem closer an agent has to reach the operation location looking for the best path and optimally avoiding collisions with other agents and then it gets operated for certain process time. As the agent actions final goal is to convert itself into a product by finding for operations on the map that convert them to a final product, So the idea is that each agent has to go through sequence of operations at process nodes and should finally reach the expedition point as finished product. This problem can be seen as scheduling problem and is classified as follows on considering resource description they are in multiple operation mode and not imposing any restrictions on tasks makes it a open shop problem and objective is the minimize the maximum makespan. However the motivation to solve our problem is to consider job shop scheduling problems, the problem was implemented on python using google OR-tools which provides us with a constraint solver. The implemented program generates an optimal sequence which represents the order of operation nodes the agents must visit to get manufactured

into some product. This sequence of operation nodes is used for path planning. The idea of flow shop to simulate assembly line works with our implementation, however flow shop simulation will be worst case because all the agents has to go through identical sequence of operations which gets worst by increasing wait time of the agents as the number of them increase.

Description of the Solver:
As already mentioned above the idea of scheduler to generate sequence of operations for ozobots is implemented using python and OR-tools. And when it comes to implementation of scheduling tasks which can be done independently from path planning. Different parts of the solver are described and here resources are assumed to be the agents or ozobots and tasks in the job description are operations at processing nodes.

Firstly, we need to import a python wrapper for or-tools constraint solver called pywrapcp which is done using below statement

```
from ortools.constraint_solver import pywrapcp
```

Next, in the main function we create the solver as

```
solver = pywrapcp.Solver()
```

In the next step we define some data about resources, processing times as list of lists and combining the both gives us job descriptions.

```
resources = [[0,1,2],[2,1,0],[1,2,0]]
processing_times = [[5,8,7],[5,8,7],[5,8,7]]
```

In the next step we calculate horizon for all the tasks as sum of processing time of each job.

```
horizon += sum(processing_times[i])
```

As next step we create jobs according to horizon calculated and considering processing time of each task in the job.

```
all_tasks[(i, j)] = solver.FixedDurationIntervalVar(horizon, processing_times[i][j])
```

In the next step we create sequence variables and add disjunctive constraints.
Disjunctive constraint is used because of the restriction that a resource can't work on two tasks at the same time.

```
disj = solver.DisjunctiveConstraint(resources_jobs, 'resource %i' % i)
all_sequences.append(disj.SequenceVar())
```

As next step we set conjunctive constraints for all jobs given.

Conjunctive constraint is used because of the condition that for any two consecutive tasks in the same job, the first must be completed before the second can be started.
In the solver conjunctive constraint is applied by built-in StartsAfterEnd() function.

```
solver.Add(all_tasks[(i, j + 1)].StartsAfterEnd(all_tasks[(i, j)]))
```

As we have already defined the resource description and restriction on the tasks in next step we set our objective which is to minimize maximum makepsan.

```
obj_var = solver.Max([all_tasks[(i, len(resources[i])-1)].EndExpr()
                      for i in all_jobs])
objective = solver.Minimize(obj_var, 1)
```

In prolog based constraint solvers as we have labeling which assigns values to the variables according to the options that are set to accomplish search, in constraint solver of OR-tools also there is something called Decision builder which creates the search tree and determines the order in which the solver searches solutions. Inside the decision builder we have solver.**Phase()** method which accomplishes search procedure.

```
vars_phase = solver.Phase([obj_var],
                          solver.CHOOSE_FIRST_UNBOUND,
                          solver.ASSIGN_MIN_VALUE)
```

Phase method has three input parameters:

- The first parameter contains the decision variables.
- The second parameter specifies how the solver chooses the next variable for the search.
- The third parameter specifies how the solver chooses the next value of the current search variable to check.

2. Path finding (Felipe Vianna)

This part of the project starts after defining the grid map and receiving the sequence of operations for each robot. The grid has the start and goal nodes, and also the definitions where the operations are, represented by color nodes in the fig 1.

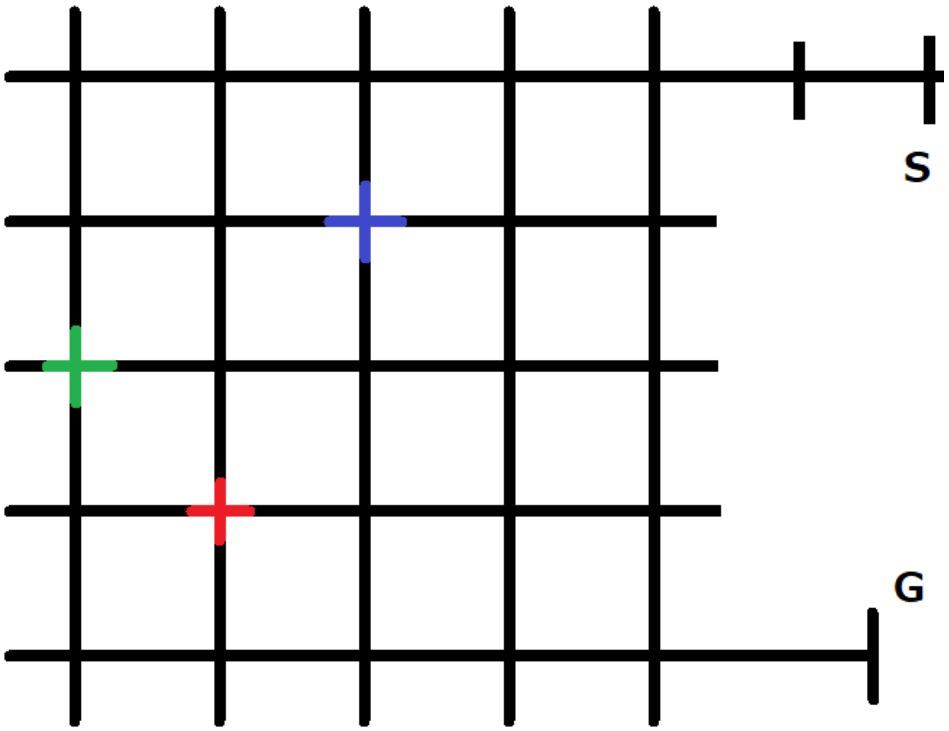

Fig 1: Grid map used for modeling the shop-floor.

To represent this grid as a graph in the program, it was used an adjacency matrix. Each node connected to each other is marked with an 1, otherwise 0. The algorithm to be used is A*, and for that reason it was easier to allow the robot to remain in same node without promoting any cycles. This means that the diagonal of adjacency matrix is 1. Also a vector represents the weights of each node, which means some nodes (the operations) hold the robot for longer time, and the others are just "pass-through" nodes. So each regular node is represent by 1 (take just 1 step to pass through it). And the operations nodes have a value according to the duration of specific operation. If a robot is passing through an operations node that but not stopping at it, it will count just 1 step, as any other non-operation node.

Each agent should visit a sequence of operations. For this reason, it was needed to find paths for each section of the full path (from start to op1, from op1 to op2, etc). *First approach* tried was to calculate several options of paths for each section of each robot sequence using **BFS** and **DFS**, ranking them from shortest to longest paths. After that, starting from best path for each robot, check for collisions. The collisions were checked comparing if 2 robots would have same node in path at the same time. If any occurs, the algorithm picked next best option for the section in which collision was detected and perform the check again. This works and found balanced paths for each robot. But it was being done for just 2 ozobots. Also the runtime of this method was getting longer, depending on how many paths were added to the ranking of paths for each robot. This greedy approach not only does not assure optimal solution but also would be very inefficient considering multiple robots.

For better generalization, allowing many operations and robots, the approach was changed. The final method was to use A* and add to the heuristics where other robots will be at each step of current robot path being searched. The heuristic returns Manhattan distance for each valid next node from current node, or infinite for invalid move. Invalid moves are the ones towards nodes that are occupied by any other robot in their current step +/- 2 steps. This bigger window of forbidden steps (-2 steps to + 2 steps) is to compensate for tight grid and difference in robots speeds. As we can't assure if one robot is going to be faster or slower than expected, it turns to be better to use previous and future steps in heuristic function. It's also invalid move to occupy operations previous robots want to go in their next steps (if they were calculated before current robot having path being processed). It's allowed by adjacency matrix that robot move to same node it is already being (equivalent to wait). Sometimes other valid moves will only make the robot farther from target, or waiting is the only accepted step.

During first tests it happened some collisions due to the fact that robots are released simultaneously but one positioned after the other. This makes their paths a little longer than the one used in calculations. To solve this, it was included in calculations one more step at start point for subsequent robots having paths calculated. This simulates the line in which they enter the process, but these additional steps added are removed after paths are found. In the real grid this additional step is already the longer distance to start node. The output of the path finding functions are the Paths for each robot, printed in text file that would be later used in OzoBlockly generator.

Path Finding Functions were all implemented in matlab, and are available in Github.
Short description of functions:

- 4 functions:
    - A_pathFinding:
        - Define adjacency matrix, weights for nodes
        - Generate paths for bots calling A_findPath, print output file
    - A_findPath:
        - Call A* to find each section path, monitoring total current number of steps used to check for heuristic
        - Concatenate all sections into the full path
    - A_star:
        - Typical A*. Best improvement step is queued.
    - A_heuristics: for each possible step, check the distance to target. If it's invalid move (node is occupied by other robot, returns inf), otherwise return manhattan distance.

3. OzoBlockly generator (Yuu Sakaguchi)

I created a Python script that generates XML which can be loaded to OzoBlockly. I first studied how OzoBlockly generates its code from XML. Once I understood how its XML should be coded, I divided the original XML code, which can be downloaded from OzoBlockly, into modules such as move forward, backward, left and right. Each action is coded with <block>, <field>, and <next> tags. It is also possible to change the moving speed, rotation degrees, waiting time, etc, by changing the parameters. Here are some example XML codes;

Code1: Follow the line to next intersection or line end
<block type="ozobot_go_to_next_intersection" id="K.!;fB9!Q-WHq-JQE%UU" x="97" y="105"></block>

Code2: Pick direction (LEFT)
<block type="ozobot_choose_way_at_intersection" id="0p|i@GQj5#Q+$0Vcz;ZF" x="162" y="201"><field name="DIRECTION">DIRECTION_LEFT</field></block>

Code3: Set line-following speed (30mm/s)
<block type="system_set_lf_speed" id="Y[tX^h)/R-]y;,RbP8m-" x="100" y="245"><value name="LF_SPEED_MMPS"><block type="math_number" id="LF1#wG:IM}iWj;rgjVo@"><field name="NUM">30</field></block></value></block>

Code4: Wait (100*10ms)
<block type="system_delay" id="lGJ7cK^{7TrhGb,);$,*" x="285" y="132"><value name="TIME_DELAY"><block type="math_number" id="25yfR1MAbwSO:9B|6f5R"><field name="NUM">100</field></block></value></block>

Additionally, you need to put a <next> tag between blocks, so that OzoBlockly can generate blocks for different actions.

We input resulting sequences of paths, generated by our MATLAB script, for each bot, then it computes which direction a bot should move at each step. The script calculates the subtraction between the current node and the next node. The possible results are 1, -1, 5, and -5. For example, if it is 1, a bot is moving one step to the East. If it is 5, a bot is moving to the South, and so on. Then, according to the current direction the robot is facing, it will compute which direction the bot should turn to. For example, if a bot is facing the North, and next node is on the West, it will turn left by 90 degrees, and so on. In the Python script, each action is composed by two blocks, change direction and follow a line. Based on the given inputs, it writes all the necessary XML blocks for the moves to

form the whole path.

The script also computes the order of bots reaching the goal based on the order of starts and the length of their paths. When the first bot reaches the goal, it will go to the end of the goal line and stop there. The second bot will follow the goal line, but it stops before crashing into the first one. And the last bot follows the same. They will park at an appropriate spot based on the order.
Each step is considered as one second, so a move from a node to another node takes approximately one second, and each waiting time is also one second. However, when a bot rotates 90 or 180 degrees, it takes the additional time, so it was difficult to set it to exactly one second. I could change the moving speed to faster when turning, but the issue usually did not cause a serious problem, so I left its speed the same.

I also tried another approach to construct the blocks. First approach was simple write XML block for each action, but the code can be too long, so I decided to store the sequence of actions into “list” in OzoBlockly, so that a bot can go back to previous node or more when it is accidentally facing a collision. I stored integers {0,1,2,3,4}, that represent more forward, back left, right, and wait. Then, I used the sensors to detect collision. If it detects another bot in front of it, and no father than one node away, it will go back to the previous node, and move back to the current node. It can back as many nodes as it avoids collisions and follow the same path to move back to the current node. However, I had some issues with this approach. First, the Ozo code became extremely confusing when I created the logic in OzoBlockly using several for-loops and if-else statements. In addition, the sensor sometime misses the collision, when it is facing another bot by 45 degrees, for example. Also, a bot can move back only when it detects a collision when it is on a node, so a bot cannot stop after started moving from a node. Otherwise, this approach worked fine, but this issue can cause more confusions at different places, so I decided not to use it.

Ozobots are quite accurate, and they do not lose its control most of the time, so we concluded that we do not need to implement the decentralized approach. As long as the given paths are correct, they do not cause collisions.

Conclusion

When considering paths with stopping points (operations) finding optimal solution becomes a hard target. In this work we tried to combine some technics for getting good sequence and good paths for robots, but couldn’t solve to get assured optimality. It was quite interesting to watch the results and the learning process during development, when we could try different methods and see how they affect the final result.