Practical Course Report

A Local Search Algorithm for Coordinated Motion Planning

Leon Füger

Date of Submission: 25. Februar 2021
Advisors: Prof. Dr. Alexander Wolff
Dr. Jonathan Klawitter

Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Zusammenfassung

In diesem Praktikumsbericht geht es um die koordinierte Bewegungsplannung einer Menge Roboter. Dabei wird ein Polynomialzeitalgorithmus vorgestellt, welcher Bewegungspläne für Instanzen findet, welche aus einem Gitter und einer Menge Startfeldern und Zielfeldern bestehen. Der vom Algorithmus erzeugte Plan liefert einen Bewegungsblauf, welcher Roboter von den Startfeldern zu den Zielfeldern befördert, dabei dürfen sich in jeder Runde mehrere Roboter um jeweils ein Feld bewegen, solange diese sich dabei nicht in die Quere kommen. Der Algorithmus versucht die *makespan*, d.h. die Anzahl der Runden bis alle Roboter auf ihren Zielfeldern sind, zu minimieren und verwendet dazu Zwischenziele für Roboter, um Blockierungen zu vermeiden, sowie eine lokale Suche, um Wege für die Roboter zu diesen Zwischenzielen zu finden. Unser Algorithmus besitzt eine Laufzeit von $O(n^2 + m \cdot k \cdot n)$, mit n der Anzahl der Roboter, m die makespan und k die Anzahl von Instanzen, welche die lokale Suche in jeder Runde durchsucht.

Contents

1 Introduction **4**

2 Problem Description **5**

3 Algorithm **6**
3.1 Local Search 6
3.2 Matching 8
3.3 Combined Algorithm 9

4 Benchmarks **11**
4.1 Tests of the algorithm 11
4.2 Calculating solutions for CG:Shop Contest 12

5 Conclusion **13**

Bibliography **14**

1 Introduction

In this practical course we considered the research question of coordinated motion planning of a set of robots. This means finding for a set of robots with start and target locations a schedule, which moves the robots from their start to their target positions. Robots can move in parallel but need to be disjoint from each other at all times. Computing an optimal schedule which minimizes the *makespan*, i.e. the total time needed for all robots to reach their goal, is strongly NP-complete as shown by Demaine et al. [DFK+19]. Our algorithm was developed specifically for the CG:SHOP 21 contest [cgs].

As with many problems of this nature there exist approximation algorithms that run in polynomial time. Masehian et al. [MS13] used a model that splits the computing of the schedule into a global heuristic for computing rough paths for each robot to its target and a local algorithm running on each robot, which provides local path planning and obstacle avoidance. Wurman et al. [WDM08] developed a system for coordinating robots in warehouses by representing the grid the robots move on as a two-dimensional weighted graph and using a standard implementation of the A^* algorithm to plan paths to storage locations and inventory stations.

Our approach uses intermediate target fields, which are spread out, to prevent robots from blocking each other, and local search through possible steps that the robots can take. The algorithm runs in $O(n^2 + m \cdot k \cdot n)$ time, with n being the number of robots an instance has, m the makespan and k the number of configurations each CPU-Core searches through in every iteration of the local search as our algorithm allows for parallel processing. The right selection of k is critical for a good compromise between makespan and running time, see Chapter 4.

2 Problem Description

In this section we formally define the problem. We have a 2-dimensional grid consisting of quadratic squares. We also have a set of n axis-aligned unit-square robots in this grid, a set $S = \{s_1, \ldots, s_n\}$ of n distinct *start* pixels (unit squares) of our grid, and a set $T = \{t_1, \ldots, t_n\}$ of n distinct *target* pixels of the grid. Each robot completely fills out exactly one square of the grid. During each unit of time, each robot can move at most one field in a direction (north, south, east or west) to an adjacent pixel, provided the robot remains disjoint from all other robots during the motions.

This condition has to be satisfied at all times, not just when robots are at pixel positions. For example, if there are robots at each of the two adjacent pixels (x, y) and $(x+1, y)$, then the robot at (x, y) can move east into position $(x+1, y)$ only if the robot at $(x + 1, y)$ moves east at the same time, so that the two robots remain in contact, during the movement, but never overlap.

The grid is not limited in any direction and we also, apart from the robots themselves, do not have obstacles. The contest [cgs] for with this algorithm was developed had instances with obstacles and instances without, we only consider obstacle free instances.

Our goal is finding a schedule that moves all robots from their starting positions S to their target positions T.

Figure 2.1 shows such an instance with the start positions in green and the target positions in red.

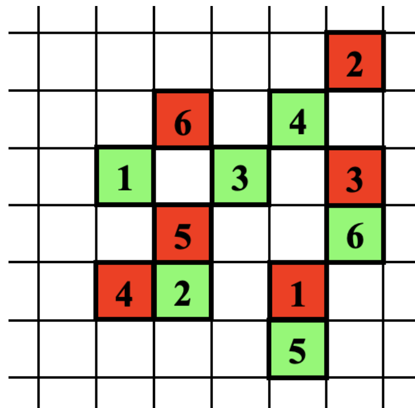

Fig. 2.1: An example of an instance with 6 robots

3 Algorithm

This section outlines our algorithm and the reasoning behind it. Section 3.1 talks about the local search which forms the basis of the algorithm which in every iteration computes sets of moves for each robot, calculates the impact each of these sets would have on the sum of the euclidean distances for each robot to its destination and simply picks the set which moves the robots closest to their destination. Section 3.2 solves a problem which causes a failure to our local search by using intermediate targets for the robots and in Section 3.3 we combine these two methods and obtain an algorithm which can solve all instances as described in Section 2 in polynomial time.

The algorithm was implemented in Python, partly because the contest for which this algorithm was developed already provided Python libraries for validating solutions and partly because of the ease of programming and libraries such as NumPy and Matlab.

3.1 Local Search

A local search is a heuristic method for solving computationally hard optimization problems. It can be used on problem that can be formulated as finding a solution among a number of candidate solutions which maximizes a certain criterion or heuristic.

The idea of our local search is the following: We have the current state of the instance, we now check in what directions any of the robots can move. To accomplish this in $O(n)$ time we use a NumPy array to store the current state of the grid. For each robot a direction can be chosen if either the specific cell is empty or occupied by a robot which has not found its target yet. We then randomly pick one direction for each robot. We get a set of moves for the robots. We then let the robots move sequentially and update the grid each time. For each robot we also have to check if the field it would be moved on was occupied by a robot at the beginning of the round. This is done by storing the direction a robot moved in at the cell it was at. This way we can check if another robot can move on that cell in the same round, because as explained in Section 2 an adjacent robot can only move in the earlier robots position if it moves in the same direction as the previous robot, to prevent overlap.

We repeat this k times and then we compare the heuristics for each of these sets of moves and pick the best one. The heuristic we use is simply the sum of the euclidean distances for each robot from its current location to its target. Euclidean distance is used instead of Manhattan distance as the former heuristic gives a better makespan on average, see Section 4.

We then execute that step and repeat the process. When a robot reaches its target, it is no longer considered in the move generation, which makes the algorithm faster towards

the end. If no robot can move, the algorithm terminates. In that case we have a valid solution. Pseudo-code 1 shows our local search.

Algorithm 1: LocalSearch(List S, List T, Int k)

Input: List of Start Fields S, List of Target Fields T, Number of Instances searched k

Output: List of Movesets M

```
1  M = []
2  curr = [evaluate(S,T), [S,T], []]
3  while true do
4      if evaluateCurr == 0 then
5          break
6      evalcurr = inf
7      nodes = []
8      for core in CPU do
9          for i = 0 to k do
10             newnode = copy(Curr)
11             grid = creategrid(newnode(S,T)
12             newmoves = generatemoves(S,T,grid) # generates moves as described
                in Section 3.1
13             step = []
14             for move in newmoves do
15                 if movevalid(move, grid) then
16                     step.append(move)
17                     moverobot(newnode[move[0] , move, grid)
18             newnode[0] = evaluate(newnode[1])
19             newnode[2] = step
20             if newnode[0] < evalcurr then
21                 nodes.append(newnode)
22                 evalcurr = newnode[0]
23     curr = smallestnode(nodes)
24     evalcurr = curr[0]
25     M.append(curr[2])
26 return M
```

The local search however starts to fail with high enough density of robots, as the test in Section 4 shows. Especially for bigger and denser instances the algorithm often causes a few robots to oscillate back and forth between 2 fields, effectively getting stuck in an endless loop.

What causes the algorithm to fail is the fact that some of the target fields are not reachable by the corresponding robot, because the way to the target is blocked by robots which have reached their target and therefore don't move anymore. The next sections outline a solution to this problem.

3.2 Matching

To solve the problem of robots not being able to access their target fields, because those fields are boxed in by robots which are already at their target locations, we use *helper fields* as intermediate targets for the robots. These helper fields are fields with even column and row number. This creates a grid in which half of the columns and half of the rows are free of targets and are passable by the robots. In other words we spread the targets out evenly so that no target field is blocked by any of the other target fields.

We obtain potential helper fields by taking the bounding box of our target fields and expanding it in x- and y- direction by doubling the side length and just taking all helper fields in that expanded bounding box. This ensures that we get no more potential helper fields than we have fields in the bounding box of our original start and target fields. We then try to find a matching between each target field and one helper field. This could be done with the starting fields as well mapping them to helper fields as the only purpose of the helper fields is to provide *intermediate targets* for our robots on their way from start to their original target.

To obtain a Matching between target and helper fields we tested 3 Methods. First, using the Kuhn-Munkres Algorithm which runs in $O(n^3)$ time and finds an optimal matching that minimizes the distances between the fields. This is done by filling a cost matrix where the rows represent the original target fields, the columns represent the helper fields and the cells of the matrix are filled with the respective distances.

Second, a simple greedy Algorithm running in $O(n^2)$ time, that assigns each target field to the nearest not yet assigned helper field. And finally just assignment in $O(1)$ time.

We tested each of these algorithms on an instance with 200 robots, measuring the sum of the (Manhattan) distances from the target fields to the helper fields. Figure 4.3 shows the results. The Kuhn-Munkres gives us a result which is about 20% better than our greedy Algorithm while the random assignment is about 2.9-times worse than the greedy matching.

Based on this we decided to use the greedy Matching, because in our estimation a 20% reduction in distance between the target and helper fields does not justify the increase

in running time by a factor of n. Pseudo-code 2 shows our greedy matching.

Algorithm 2: Matching(List T)

```
Input: List of Target Fields T
Output: Sorted Sublist of Helper Fields R
1  B = ComputeBoundingBox(T)
2  doubleXandYSize(B)
3  H = EvenFields(B)
4  R = []
5  for t in T do
6      currdist = inf
7      currhelp = 0
8      for h in H do
9          dist = distance(t, h)
10         if dist < currdist then
11             currdist = dist
12             currhelp = h
13     R.append(currhelp)
14     H.remove(currhelp)
15 return R
```

3.3 Combined Algorithm

With our new helper targets we can now use our local search to *always* find a schedule for our robots from their starting locations to the helper targets, since no target field is blocked by any other occupied target field.
However we still need to get the robots to their original destination. To accomplish this we *use the original target fields as our starting locations* and the helper fields as our targets. We then again execute our local search and we will find a schedule that moves robots from the original targets to the helper fields. Note that the second local search is obviously not executed on the same instance but on a new one with new robots starting at the target locations of the robots of the original instance.
What we can now do since in both instances the robots end up on the same helper fields, we can reverse the order of sets of moves in the second schedule and for each moved robot flip the Direction it was moved in (i.e. north to south , east to west etc.). We can now add this modified second schedule to the first schedule and the result is a schedule that moves the robots from the original starting locations to the original target locations.

Figure 3.1 shows an instance in the first local search. Robots that have found their target (helper) field are green, those that are still moving are shown in blue and not yet occupied target fields are shown in red. Note that the red helper fields are spread out enough so that each is accessible.

Our algorithm always terminates with a valid solution, because all of the target fields can be accessed by the specific robot and when a robot has found its target it is no longer

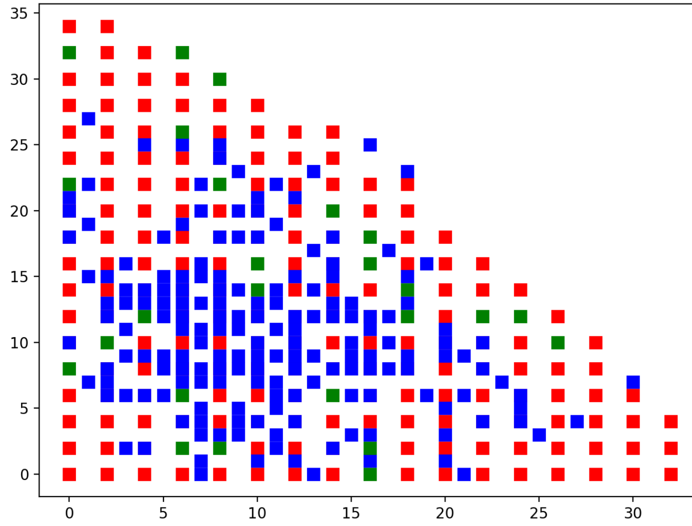

Fig. 3.1: An instance with 200 robots being solved

considered in the move generation. This causes the number of moving robots to decline until all have found their target. The Pseudo-code 3 shows how our two algorithms are combined to produce a valid solution.

Algorithm 3: CombindedAlgorithm(List S, List T, Int k)

Input: List of Start Fields S, List of Target Fields T, Number of Instances searched k

Output: Solution R

```
1 H = Matching(T)
2 sol1 = LocalSearch(S, H, k)
3 sol2 = LocalSearch(T, H, k)
4 sol2.reverse()
5 for sets in sol2 do
6     for move in sets do
7         FlipDirectionOfMove(move)
8 R = sol1 + sol2
9 return R
```

4 Benchmarks

4.1 Tests of the algorithm

The benchmarks were done on a Machine with an 4-core Intel i5-6600K processor, 16 GB RAM and running Windows 10.

Euclidean or Manhattan Distance as our Heuristic. We tested two instances, one with 50 and one with 200 robots. We ran each instance 10 times, half of these runs with Manhattan and Euclidean distance as heuristic respectively. In each iteration the local search searched through 20 possible move-sets. Figure 4.1 shows the average makespan for each of these setups. Based on this we chose Euclidean Distance as the heuristic for our local search.

n	Manhattan	Euclidean
50	185	161
200	787	656

Fig. 4.1: The makespan gets 15% better by using Euclidean Distance as Heuristic

Likelihood of local search failing. Here we took a grid of size 10 by 10, filled it with an increasing number of robots with random non overlapping start and target positions and executed our local search. We did test every configuration 10 times. Figure 4.2 shows the results. At 16 robots the algorithm starts to fail with some robots not being able to access their target field.

n	likelihood of the algorithm terminating
2	100 %
4	100 %
8	100 %
16	80 %
32	0 %

Fig. 4.2: The algorithm starts to fail with 16 robots

Test of the matching algorithms. We tested our three different matching algorithms on an instance with 200 robots. As expected simple random assignment gives a large total distance. We decided on the greedy method simply because of the significantly lower running time.

Algorithm	Sum of the distances	Running time
Kuhn-Munkres	1242	$O(n^3)$
Greedy	1538	$O(n^2)$
Random Assignment	4440	$O(1)$

Fig. 4.3: Testing the different Matching algorithms on an instance with 200 robots.

Benchmark of final algorithm. We tested an instance with 50 robots. This instance was tested with four different values for k, i.e. the amount of moves we search through in each iteration of the local search. We test for $k = 1$, and for 10, 100 and 1000 and since we had 4 CPU Cores this meant searching through four times that many instances in each test. Each of these configurations were run 5 times and the average of the running time and the makespan were taken. A lower bound for the makespan can be computed simply by taking the biggest Manhattan distance between all start and target fields. For this instance we get $OPT \geq 15$. Figure 4.4 shows the results. Generally, keeping the number of instances searched through between 10 and 100 seemed to us to be a good compromise and we applied this to our calculations for the CG:SHOP contest.

k	makespan	running time in seconds	makespan * running time
1	317	2.80	888
10	130	3.62	471
100	103	17.08	1833
1000	89	81.39	7244

Fig. 4.4: Test for different values of k on an instance with 50 robots

4.2 Calculating solutions for CG:Shop Contest

To compute schedules for the CG:SHOP 21 [cgs] contest we used the High Performance Computing Cluster of the University of Würzburg. Taking the 16-Core Server CPU into account the algorithm searched through 48 instances in each iteration and, over the span of two weeks, computed solutions for all instances without obstacles and up to and excluding 5000 robots. It also computed a solution for the biggest instance provided by the contest with 9000 robots in 32 hours.

5 Conclusion

The algorithm described in this report finds schedules for coordinated motion planning of a set of robots. It can solve all instances that fit the definition in Section 2 in polynomial time by splitting the problem into two instances with spaced out target fields and solving them by using a local search algorithm. The local search part of the algorithm could probably be further improved by using more elaborate move generation and heuristics for finding the best move set. However, generalizing the algorithm to also work on instances that have obstacles would likely require major modifications to both matching and local search. Furthermore we have to consider the running time of $O(n^2 + m \cdot k \cdot n)$ where the first part (n^2) corresponds to the matching algorithm. This part is responsible for only a minuscule fraction of the total running time, which means that the total running time of the algorithm significantly exceeds quadratic time even for very low k. As such the algorithm works best with instances which have a discrete number of robots.

Bibliography

[cgs] https://cgshop.ibr.cs.tu-bs.de/competition/cg-shop-2021/#problem-description.

[DFK+19] Erik D Demaine, Sándor P Fekete, Phillip Keldenich, Henk Meijer, and Christian Scheffer: Coordinated motion planning: Reconfiguring a swarm of labeled robots with bounded stretch. *SIAM Journal on Computing*, 48(6):1727–1762, 2019.

[MS13] Ellips Masehian and Davoud Sedighizadeh: An improved particle swarm optimization method for motion planning of multiple robots. In *Distributed autonomous robotic systems*, pages 175–188. Springer, 2013.

[WDM08] Peter R Wurman, Raffaello D'Andrea, and Mick Mountz: Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1):9–9, 2008.

Erklärung

Hiermit versichere ich die vorliegende Abschlussarbeit selbstständig verfasst zu haben, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben, und die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde unter Erlangung eines akademischen Grades vorgelegt zu haben.

Würzburg, den 25. Februar 2021

. . . .

Leon Füger