# Coordinated Motion Planning

**Yunhao Yu**
Ecole Polytechnique
yunhao.yu@polytechnique.edu

**Jingtian Ji**
Ecole Polytechnique
jingtian.ji@polytechnique.edu

## 1 Introduction

Coordinated motion planning has always been one of the critical research areas of computational geometry. In recent years, various excellent algorithms have emerged to solve the multiagent planning problem. In a discrete environment like the grid world, many algorithms have requirements on robots' movement, such as the number of robots, and the distribution of obstacles. However, in the real worlds, such as autonomous vehicles and logistics and warehouse robots, motion, environment constraints, and uncertainty are more restraint and essential, so in this paper, we will try to propose a set of efficient offline planning algorithms that are suitable for a large number of robots in complex terrain with the presence of obstacles and low separability.

This paper's contribution involves three parts: (1) Based on the greedy and compromise mechanism. We realize local mutual communication between the agents. (2) Use Metropolis Hasting simulated annealing algorithm to approximate the local optimum at each step according to agents' potential behavior. (3) For each feasible solution, we propose an optimization process in terms of makespan and total distance, which reduces the vibration phenomenon caused by robots conflicting with each other or avoiding deadlock and enhances the parallel movement of non-conflicting robots. We will also give theoretical proofs of our optimization algorithms and compare the performance on problems of different scales in the paper.

## 2 Preliminaries

Let the graph $G = (V, E)$ be a simple, undirected, connected graph. Suppose $|V| = N1 \times N2$ and $O = \{o_1, o_2, , o_l\}$, is a set of obstacles. The robot cannot occupy the obstacle node at any time. The robot can only move in four directions, cannot overlap, rotate, or exchange positions. Particularly, 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. In the moving process, at a given time t, each robot must occupy a node. We use configuration $X_t$ to indicate the mapping of a robot index to a node: $(1, 2, n) \to V$ and we use $x_i^t$ to designate the node where $i$th robot is on at step $t$. If all robots from time $t$ to time $t + 1$ satisfy the above constraints, we call $P_t = (X_t, X_{t+1})$ a transition, then given $S = s_1, s_2, , s_n$ n departure nodes, and $T = t_1, t_2, , t_n$ n target nodes. our final set of feasible solutions is $Sol = (X_S, X_1, \ldots, X_T)$. For each $S$ and $T$, we can point out that the potential smallest makespan = $max(d(s_i, t_i), i \in \{1, 2, , n\})$, the potential smallest moving distance = $\sum_{i=1}^{i=n} d(s_i, t_i)$.

It is worth noting that we will encounter many difficulties and constraints, unlike the other papers' problem settings[1][2]. (1) Curse of dimensionality: Enumerate all actions under each step is $O(5^n)$, it is difficult to find the shortest distance or the shortest time. (2) Constraints: There are many obstacles and moving constraints between robots, requiring cooperation between agents. For example, the agent who arrives at the destination first may block other agents' way, which requires the agent to show appropriate altruistic behaviour.

Preprint. Under review.

# 3 Algorithm Description

## 3.1 Greedy search and Compromised behavior

### 3.1.1 Greedy search

According to the given initial position $S$, the target position $T$ and the obstacle $O$, we can delineate a boundary for the grid world and calculate the all pairs' shortest distance in the grid while ignoring all robots' existence. With the help of the calculated distance, we can define the robot's greedy operation: the next node selected each time is the node that can reduce the distance between itself and the target node: $x_i^{t+1} = argmin(d(v, s_i), v \in Neighbor(x_i^t))$. Due to the emergence of other robots, the greedy selection is not always feasible.

### 3.1.2 Compromised behavior

For the following notation, we omit the index of time $t$, and all the comparisons are made at the same step $t$. We introduce the compromise judgment to avoid conflicts between robots: The robot will make the compromise judgment if there other robots close enough to itself. Here we define the distance threshold as 3. When $d(x_1, x_2) \leq 3$, the greedy search will switch to compromise judgment.

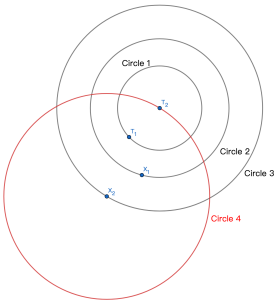


Figure 1: Sequence Problem

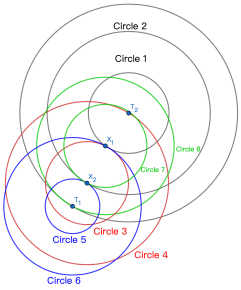


Figure 2: Collision Problem

**Sequence Problem** Robot 1 and robot 2 have a sequence conflict, and robot 1 needs to compromise is equivalent to:

1 Robot 2 is further away from the destination: $d(x_2, t_2) > d(x_1, t_1)$;

2 The destination of robot 1 is on the way between robot 2 and its destination: $d(t_1, t_2) < d(x_2, t_2)$; $d(x_2, t_1) < d(x_2, t_2)$;

3 Robot 1 is currently on the way between Robot 2 and its destination: $d(x_1, t_2) < d(x_2, t_2)$; $d(x_1, x_2) < d(x_2, t_2)$

We can describe this problem graphically (Fig.1), the five inequalities can be interpreted as Circle 1 is inside Circle 3, Circle 2 is inside Circle 3, $t_1$ is inside Circle 4, $x_1$ is inside Circle 4 and $d(x_2, t_2) > d(x_1, t_1)$.

Satisfying the above three conditions, we set the robot one into a compromised state. The former will perform a compromise operation, and the latter will perform a greedy operation.

**Collision Problem** Robot 1 and robot 2 collide with each other, and robot 1 needs to compromise is equivalent to:

1 Robot 1 is currently on the way between Robot 2 and its destination: $d(x_1, t_2) < d(x_2, t_2)$; $d(x_1, x_2) < d(x_2, t_2)$

2 Robot 2 is currently on the way between Robot 1 and its destination: $d(x_2, t_1) < d(x_1, t_1)$; $d(x_2, x_1) < d(x_1, t_1)$

Similarly, we can describe these constraints graphically:

Satisfying the above three conditions, we set the robot 1 into a compromised state. The former will perform a compromise operation, and the latter will perform a greedy operation.

The compromise operation will be performed by assigning more probability on the direction, enabling the compromised agent to move far away from the other agent and its target node. The final movement will be sampled according to the probability.

### 3.2 Global search

The greedy and compromise search mentioned above refers to the action selection at a given time. How to form a set of feasible transition solutions through the selection of a single robot?

First of all, we give an order to whole robots, according to this order to each robot uses greedy and compromise search, the robot that makes the judgment first will not consider the current position and next position of the subsequent robot. It will leave a direction mark on its current node and the next node selected. In addition to greedy and compromise operations, the subsequent robot's next node must not overlap with others who have already made their choice, and its direction mark must not conflict with the existing direction mark.

Using a greedy and compromise search in a given order does not guarantee a transition solution, but it must be a feasible transition solution if the selection is successful. Since different execution orders will bring different results, we want to find the optimal execution order and transition solution in the current state. Therefore, we adopted the MCMC-based simulated annealing. In order to use the simulated annealing, we need a value function. Here we define $Value(X_t) = \sum_{i=1}^{n} d(x_i^t, t_i) - \sum_{i \in compromised\ state} \alpha_i$. This value function encourages all robots to move towards the target location and make compromises. By constantly trying different orders, we can get a more reliable transition solution at each step; the specific execution process is as follows.

**Input :** Current State $X_t$, initialized $Order$
**for** $i = 1 \dots N$ **do**
  $Order'$ = Propose a feasible Order;
  ${X_{t+1}}'$ = $Greadу\ and\ compromise\ search\ (Order', X_t)$;
  $V' = Value({X_{t+1}}')$;
  Acceptance Probability $h = min(1, \frac{exp(-\frac{V}{T_i})}{exp(-\frac{V'}{T_i})})$;
  **if** *accept* **then**
    $V = V'$;
    $Order = Order'$;
    **if** $V > Best\ V$ **then**
      $Best\ X_{t+1} = X_{t+1}$;
      $Best\ V = V$;
      $Best\ Order = Order$;
    **end**
  **end**
**end**
**Output :** $Best\ X_{t+1}$

**Algorithm 1:** Simulated annealing

### 3.3 Optimization

Since we have introduced some randomness in our global search algorithm, thus we observed that the solutions we obtained after the global search have two potential problems :

1 The positions for some agents will oscillate.

2 The moves for several agents may be merged into one single move.

Thus, we devise two algorithms to solve these two problems, these two algorithms can also be used to optimize feasible solutions to the original problem.

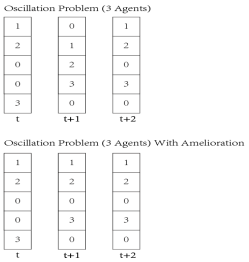


Figure 3: Oscillation Problem

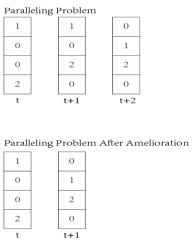


Figure 4: Paralleling Problem

### 3.3.1 Algorithm 1: Rewind2Stabilize

The first algorithm is used to solve the oscillations problems. We define **an agent $i$ has an oscillation problem** if the its moves $(x_i^t)_{t\in\{1,\dots,T\}}$ ($T$ is the total make span) satisfies the following condition:$\exists t_1, t_2 \in \{1,\dots,T\}, t_1 < t_2$ that makes $x_i^{t_1} = x_i^{t_2}$ and if we only change this agent's move in the original feasible solution to $\forall t \in \{t_1,\dots,t_2\} x_i^t = x_i^{t_1}$ and this new solution is also feasible.

For this algorithm, we use two inputs: **nodeHistory, stateHistory**. The former use an HashMap to record the history (time and agent index on this node) of a Node, thus we can use O(1) to get, add or remove an item, and O(n) to find a record before $t$. The latter uses a simple array to record the Node for every agent for every $t$. Thus, we can get or modify the position of an agent at time $t$ in $O(1)$.

This algorithm is composed with 2 parts.

**Part 1: Find a pattern**

We define a **Node $v$ has a pattern at time t** if this Node $v \in V$ satisfy the following three conditions:

1 The robot $i$ appears at this Node at time $t$ : $x_i^t = v$

2 $\exists t_0 \in \{0,\dots,t-2\}$ that $x_i^{t_0} = x_i^t = v$

3 $\forall t_1 \in \{t_0+1,\dots,t-1\}$ no agent is on this node $v$.

And we call **the corresponding agent $i$ has a pattern a pattern between $t_0$ and $t$.** The purpose is to find out whether a pattern has appeared on any Node at time t. Since at time $t$, the first conditions is satisfied if and only if the Node $v \in \{x_i^t, \forall i \in \{1,\dots,n\}\}$, thus we can start by traversing all the active agent at time $t$.

**Property** If a Node has a pattern at time $t$, then at $t-1$ there is no agent on this Node.

**Proof** Suppose $t_0 \le t-2$, and the Node has a pattern between $t_0$ an $t$. Then according to the third condition, $\forall t_0 + 1 \le t_1 \le t-1$ there is no agent on this Node. We can choose $t_1 = t-1$. □

**Part 2: Stabilize**

Since we only use 2 data, once we find a pattern, in other words, we find an Node who satisfies the following situation:

1 The agent $i$ is at this Node at time $t_0$ and $t$, $t - t_0 > 1$.

2 $\forall t_1 \in \{t_0+1,\dots,t-1\}$ this Node is empty, no agent is on this Node.

Then we should try to update the original solution by performing 2 types of operations relating to the agent $i$ who has a pattern between $t_0$ and $t$:

1 $\forall t_1 \in \{t_0+1,\dots,t-1\}$ We remove the history Node $x_i^{t_1}$ of agent $i$.

2 $\forall t_1 \in \{t_0+1,\dots,t-1\}$ We set $x_i^{t_1} := x_i^t = x_i^{t_0}$

**Input :** stateHistory,nodeHistory,t
**Output :** The index of the agent who has a pattern between $t0$ and $t$, $t0$, $t$

*We traverse all the active agent at given $t$;*
**for** *$i$ in agentIndex* **do**
    currentNode ← The Node which agent $i$ is in at time t;
    lastRecord ← The fist record of currentNode before time t;
    // lastRecord is happened at time $t_0$, no agent is on this Node between $t_0 + 1$ and $t - 1$ ;
    **if** *lastRecord verifies the second and third definiton of Pattern* **then**
        return i,$t0$,$t$;
    **end**
**end**
We can't find a pattern at time t;
return Null;

**Algorithm 2:** Find a pattern

*We modify the history in-place.*;
**Input :** stateHistory,nodeHistory,agentIndex $i$,$t_0$,$t$
**Output :** None

targetNode ← The Node which agent $i$ is in at time t;
**for** $T \leftarrow t_0 + 1$ **to** $t - 1$ **do**
    wrongNode ← The Node which agent $i$ is in at time T;
    stateHistory[T][i] ← targetNode;
    We remove the fact that agent $i$ didn't come to wrongNode at time T in nodeHistory.;
    We add the fact that agent $i$ came to targetNode at time T in nodeHistory.;
**end**

**Algorithm 3:** Stabilize

**The Pseudo code for Rewind2Stabilize**

By using these two components, we have established our Replay2Stabilize algorithm which rewind our feasible solution obtained after the global search. This algorithm will remove all the oscillation in our feasible solution.

**Input :** stateHistory,nodeHistory
**Output :** The stabilized version of stateHistory,nodeHistory

Initialization;
$t \leftarrow$ The total make span;
**while** *$t > 0$* **do**
    res ← FindAPattern(stateHistory,nodeHistory);
    // For each $t$, we will keep optimizing until no new pattern appears;
    **while** *res!=Null* **do**
        //We have successfully find a pattern, res contains the corresponding agent index, $t_0$ and $t$;
        Stabilize(stateHistory,nodeHistory,res);
        res ← FindAPattern(stateHistory,nodeHistory);
    **end**
    t-=1;
**end**

**Algorithm 4:** Rewind2Stabilize

**Lemma** This algorithm terminates.

**Proof** We only look at the inner loop, because if the inner loop terminates, then we have $t := t - 1$, which will eventually conflict to the outer loop and finally terminates. Thus, we want to know whether we will always find a pattern at time $t$. We can use contradiction to prove. For a given $t$, if there always exists an agent $i \in \{1, \ldots, n\}$ who will always have a pattern between $t_0$ and $t$, we note

$x_i^t = x_i^{t_0}$ its associated Node. Then according to the property of **pattern**, the fact that there always exists a pattern could be translate into the Node $x_i^t$ is always empty at time $t-1$. This is not correct, since every time we find a pattern, we will try to stabilize it, which will cause the Node $x_i^t$ is occupied by the agent $i$ at time $t-1$, thus we have a contradiction. □

This proof also gives the following corollary.

**Corollary** At given $t$, if we find agent $i$ has a oscillation problem, then after we stabilize it, this agent will have no more oscillation problem at $t$.

**Lemma** This algorithm will generate a feasible solution, in other words, there is no collision.

**Proof** Since only the part stabilize modifies the original solution, thus we only need to prove that every change is feasible. This is trivial, because every time we find a pattern at Node between $t_0$ and $t$, we are sure this Node remains empty between $t_0+1$ and $t-1$. Thus, if we demand the corresponding agent to stay on this Node during $t_0+1$ and $t-1$, this won't cause any collision, since there is no agent trespassing this Node.□

**Lemma** This algorithm will render a solution without any problem of oscillation.

**Proof** In fact if there is a problem of oscillation, then we will always find a pattern. Thus, the loop invariant *res!=Null* means that even if after solving the current there might emerges a new oscillation problem for another agent, we will continue solve this new problem until there is no more problem of oscillation that between $t_0 \in \{1,\dots,t-1\}$ and $t$. Finally, since we are certain there is no more oscillation problem that ends at $t, \forall t \in \{1,\dots,T\}$ ($T$ is total make span), this algorithm will render a solution without any problem of oscillation. □

### 3.3.2 Algorithm 2: Rewind2Parallelize

The second algorithm is used to merge the moves of several agents in to one single move so as to shorten the total make span of our solution. To do this, we introduce the following notation. We call an **agent** $i$ **is active at time** $t$ if this agent changes its position between $t$ and $t+1$, we note its move $a_i^t = (x_i^t, x_i^{t+1})$. We call an **Active Group at time** $t$ $A_t$ a set containing all the moves of active agent at time $t$. This algorithm consists of two parts, in the first part, we find if we can arrange an action of an agent in advance, if we can then we use the method in the second part to arrange this action in the final solution.

**Part 1: Find previous active moment**

The purpose is to find $t^*$ that we can arrange an agent's move in advance from $a_i^t$ to $t^*, t^* < t$.

**Input :** agentIndex $i$,$t$
**Output :** $t^*$

previousAction ← The previous action $a_i^{t_1}$ before the current action $a_i^t$;
$t_1$ ← The time when the previous action took place ;
$s$ ← The start Node of the current action $a_i^t$ ;
$e$ ← The end Node of the current action $a_i^t$ ;
**for** $T \leftarrow t_1+1$ **to** $t-1$ **do**
  **if** $e$ *is occupied by another agent at time* $T$ **then**
    $t^* \leftarrow T+1$;
  **end**
**end**
**if** $t^* < t$ **then**
  return $t^*$;
**else**
  return -1;
**end**

**Algorithm 5:** Find Previous Active Moment

**Lemma** This algorithm is correct, we can arrange the agent's move $a_i^t$ to $a_i^{t^*}, t^* < t$ without causing any collision.

**Proof** Since we only consider the move of one single agent $i$, we only need to look at the two Nodes to prove the correctness of this algorithm: the start Node $x_i^t$ and the end Node $x_i^{t+1}$ of the move $a_i^t$ at time $t$. We note $a_i^{t_1}$ is the previous move for the agent $i$ before $a_i^t$.

For the start Node $x_i^t$, since the original solution is feasible, the agent $i$ stays at Node $x_i^t$ between $t_1 + 1$ and $t$, thus if we want the agent $i$ to leave the start Node at $t_1 + 1 \leq t^* \leq t - 1$, this will not result in a collision at the start Node.

For the end Node $x_i^{t+1}$, we make sure that this Node is Empty at the beginning of time $t^*$ before the agent $t$ enters the end Node. For instance, if the last time another agent $j$ occupies the end Node is at time $T$, then $j$ will leave this end Node at the end of the time $T$, which will leaves the end Node empty after $T + 1$.

Thus, our algorithm is correct and we can obtain a solution without causing any collisions.

**Part 2: Parallelize**

Since we have found out the move of agent $i$ at time $t$ could be merged in a previous action at time $t^*$, thus the purpose of this part is to merge these two actions. We use an Array List *originalNodes* and an Array *activeGroups*. The former allows us to get or modify the positions for every agent at time t in $O(T)$ (T is the total make span). The latter allows us to get or modify the actions at time $t$ in $O(1)$.

**Input :** originalNodes,activeGroups,agentIndex $i$, $t$,$t^*$
**Output :** None

targetNode $\leftarrow$ The destination Node of the agent $i$ at $t$;
activeGroups[$t$].remove(action of agent i at time $t$);
activeGroups[$t^*$].add(action of agent i at time $t$) ;
**for** $T \leftarrow t^*$ **to** $t - 1$ **do**
    We update the new position of agent i at time T;
    originalNodes[T][i] $\leftarrow$ targetNode;
**end**

**Algorithm 6:** Pseudo code of Parallelize

**Pseudo code of Rewind2Parallelize**

**Input :** agentIndex $i$,$t$
**Output :** None

$t \leftarrow$ The total make span;
**while** $t > 0$ **do**
    **for** $i$ *in activeAgentIndex* **do**
        $t^* \leftarrow$ FindPreviousActiveMoment(i,t);
        **if** $t^*$*!=-1* **then**
            Parallelize($i$,$t^*$);
        **end**
    **end**
    t-=1;
**end**

**Algorithm 7:** Pseudo code of Rewind2Parallelize

Finally, we can reconstruct the solution by skipping the moment where no agent is active.

**Lemma** The algorithm Rewind2Parallelize terminates and generate a feasible solution.

**Proof** To prove that this algorithm terminates, we only need to prove that once we have found two actions that could be merged into one by the function *FindPreviousActiveMoment*, and after we have used the function of *Parrallelize* to merge them, these two actions won't ever be detected by the function *FindPreviousActiveMoment* at time $t$. This is true, because after the *Parallellize*, we have removed the agent from the active agent index at time $t$. Thus the algorithm terminates. This algorithm produces a feasible solution because of the Lemma of the function *FindPreviousActiveMoment*, which

indicates that once the function returns a $t^*! = 1$ then the merge process is reasonable. Thus when the algorithm terminates, it gives us a feasible solution. □

# 4 Algorithm Analysis

In this section, we try to analyze the complexity for the algorithm described in the sections above.

## 4.1 Time Complexity of the algorithm Rewind2Stabilize

We denote $T$ the total make span, $n$ the number of total agents. Then the time complexity in the worst case of the algorithm Rewind2Stabilize is $O(n^2 * T^2)$.

**Proof** By using the Corollary, we know that for a given $t$ we only run the *FindAPattern* $n + 1$ times and *Stabilize* $n$ times.

Furthermore, the *FindAPattern* has a worst complexity of $O(n * T)$. The term $T$ is caused by looking the *lastRecord* in a Hash Map, in the case when this Node is always occupied by an agent. But if we have lots of the position where agent can move, then this term could be amortized into $O(1)$, which gives the algorithm a runtime of $O(n)$ in the normal situation. This can also be ameliorated to $O(1)$ if we store the unchecked agent index into a set instead of using a for loop.

Besides the *Stabilize* has a worst time complexity of $O(T)$, since checking or removing an element in a Hash Map costs $O(1)$. But normally, this term could be amortized into $O(1)$, since we can consider the difference between two active moment $t - t_0$ on a Node to be a constant.

Thus, for a given $t$, in the worst case the algorithm has a time complexity of $O((n+1)*n*T+n*T) = O(n^2 * T)$, but with a little bit amelioration and in the normal situation, the time complexity is $O((n + 1) * 1 + n * 1) = O(n)$.

Finally, since we check for every $t \in \{1, \ldots, T\}$, the total time complexity of the algorithm in the worst case is $O(n^2 * T^2)$, and with a little bit amelioration and in the normal situation, the time complexity is $O(n * T)$.

## 4.2 Space Complexity of the algorithm Rewind2Stabilize

We denote $T$ the total make span, $n$ the number of total agents. Then the space complexity of the algorithm Rewind2Stabilize is $O(n * T)$.

**Proof** Since our algorithm is an in place algorithm, the major cost of space comes from the storage of *stateHistory* and *NodeHistory*. Since we have a defined position for the agent at every moment, the *stateHistory* uses $O(n * T)$. Since we use a Hash Map to store the *NodeHistory*, we don't store redundant elements. In this case, we should store the same number of elements as the *stateHistory*, thus *NodeHistory* uses also $O(n * T)$. Finally, we have the space complexity of the algorithm Rewind2Stabilize is $O(n * T)$.

## 4.3 Time Complexity of the algorithm Rewind2Parallelize

We denote $T$ the total make span, $n$ the number of total agents. Then the time complexity of the algorithm Rewind2Parallelize is $O(T^2 + n * T)$.

**Proof** At every $t$, in the worst case, we have $n$ active agent. Thus at $t$, we do $n$ times *FindPreviousActiveMoment*. In the worst case, the previous active moment is at $t = 1$, thus, *FindPreviousActiveMoment* has a time complexity of $O(T)$. Then the *FindPreviousActiveMoment* totally uses $O(T^2)$. As for *Parallelize*, we can neglect the two operations related to *activeGroups* since they can be done in $O(1)$. Since the $t$ is decreasing in the algorithm Rewind2Parallelize, thus for each agent, we change its position in originalNodes for at most $O(T)$. Given the fact that, if an agent's move is changed to $t^*$ , then it will only be regarded as active once the current time $t$ descends to $t^*$. Thus, for $n$ agent, the *Parallelize* takes $O(n * T)$. Finally we spent $O(T)$ to reconstruct the new solution. Thus, the time complexity of the algorithm Rewind2Parrallelize is $O(T^2 + n * T + T) = O(T^2 + n * T)$.

Table 1: Experiments Results

| Name | N of robots | Map Size | Density | Original | Optm 1 | Optm 1+2 | Theoretic |
|---|---|---|---|---|---|---|---|
| small 000 | 10 | 15 * 15(27) | 0.06 | 45(190) | 43(116) | 28(116) | 19(100) |
| small 001 | 30 | 15 * 15(20) | 0.17 | 97(1020) | 78(456) | 64(456) | 17(246) |
| small free 000 | 30 | 15 * 15(0) | 0.15 | 71(709) | 51(267) | 34(267) | 13(187) |
| small free 001 | 40 | 15 * 15(0) | 0.2 | 103(1574) | 87(1532) | 82(616) | 13(266) |
| galaxy cluster | 80 | 25 * 25(0) | 0.14 | 126(3357) | 106(1579) | 98(1579) | 27(845) |
| buffalo 000 | 63 | 30 * 30(98) | 0.12 | 174(4181) | 160(2235) | 151(2235) | 54(1547) |

### 4.4 Space Complexity of the algorithm Rewind2Parallelize

We denote $T$ the total make span, $n$ the number of total agents. Then the space complexity of the algorithm Rewind2Parallelize is $O(n * T)$.

**Proof** The main space cost comes from the *originalNodes* and *activeGroups*. A similar proof to the previous part discussing the Space Complexity of the first optimization algorithm shows that *originalNodes* uses $O(n * T)$ . as for the *activeGroups*, in the worst circumstance where all the agent move at everytime, it uses $O(n * T)$. Thus the total space complexity of the algorithm Rewind2Parrallelize is $O(n * T)$.

## 5 Experiments

In our experimental part, we conduct a comparative analysis for different inputs, study the original feasible solution, compare the results after two optimizations, and then compare it with the theoretically optimal solution. It is not difficult to see that our optimized solution is perfect. First, our density is the ratio of the number of robots to the number of reachable nodes. All solutions are in the form of makespan (total distance). The baseline value we pointed out refers to the minimum value mentioned in the premise.

The result of our initial solution is imprecise. However, the first optimization can reduce half of the total distance of the initial result, and the degree of approximation to the optimal solution depends on the density of the map. The lower the density, the closer the optimization result is to the optimal solution. Our most intuitive explanation for this is that the higher the density, the easier it is to produce motion conflicts. The baseline solution we proposed may be unreachable because of the motion constraints, so our approximation is satisfying and our results are usually the 3-approximation solutions.

## 6 Conclusion

In this work, we propose a planning algorithm for multi-robots with many obstacles and low separability. Compromise and greedy search can effectively achieve local communication and fast movement (after initialization, the single choice is O(1)). The simulated annealing algorithm quickly finds the local optimal for transition solution efficiently, avoiding brute force search resulting in an exponential complexity. Our two sets of general optimization algorithms effectively eliminate all oscillating behaviours, improve the robot's parallelism, and significantly optimize the moving distance and makespan under strict proof guarantees. It is worth mentioning that our optimization algorithm is not limited to optimizing our feasible solutions, and its ideas can apply to any feasible solutions of a multi-robots planning problem to eliminate the problems of oscillation and parallelization.

In future work, inspired by reinforcement learning, we will adopt the try-and-error idea. (1) We can parameterize the value function and continuously optimize it by lots of realizations. (2) We can combine the annealing algorithm and greedy search into a strategy network, and adjust the probability of greedy and compromise behaviour according to the value function. (3) We can set different rewards (penalties) such as moving distance or makespan, to achieve different purpose-oriented reinforcement learning to get different scoring functions.

## References


[1] 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.

[2] Jingjin Yu. Constant factor time optimal multi-robot routing on high-dimensional grids in mostly sub-quadratic time, 2018.