AI HW2: Mazeworld Solution

Tianyuan Zhang

September 28, 2016

1 Introduction

The Maze Problem is one of the most classic and interesting problems in the artificial intelligence field. It is mainly about searching the branch through several possible states. In this report, I majorly discuss three parts, 1) A* algorithm as a searching method; 2) Multirobot problem, where the collision problem is introduced 3) Blind robot problem, where the robot doesn't know where it starts and 4) finally, some further discussion and paper review.

2 Build the Maze

2.1 Maze Representation

I use the plain txt file as the input file and use characters to represent the maze - '#' represents the wall and '.' represents the floor. Here is a sample 7*7 maze:

```
.......
.##....
..##...
.......
..##...
#.###..
....##.
```

2.2 Coordinate

For convenient, I use coordinate (0, 0) represents the top-left corner position and (m, n) represents the bot-right corner position, where m = width - 1 and n = height - 1.

2.3 Read a Maze

There is a class named MazeWorld to read the maze and provide some basic functions for the maze. Here is an example to read a maze from a file named *simpleMaze.txt*:

```
1 MazeWorld maze = new MazeWorld("singleMaze.txt");
```

Note that the input file is under the folder of the project.

3 A* search

3.1 Basic Idea

A* search is a type of informed search, that it knows some information about the current position and the goal, which is different from BFS. In BFS, the search only considers the cost (The distance from the start to the current position), while in A* search, the heuristic is also introduced. A heuristic estimates a distance from the current state to the goal state. Hence we can regard the *cost* as the estimation for the past and *heuristic* as the estimation for the future. Every time when A* search need to pick next state, it will always find the highest priority one in the priorityQuene and search through it.

3.2 Single Robot Problem

Given a start position and a goal, find a path. Naturally, the state in this problem is the position - the coordinate (x, y) of the robot.

3.3 Code Implementation

```
public List<SearchNode> astarSearch(){
    resetStats();

    PriorityQueue<SearchNode> my_pq = new PriorityQueue<SearchNode>();
    HashMap<SearchNode, Double> visited =
      new HashMap<SearchNode, Double>();
    HashMap<SearchNode, SearchNode> parent =
      new HashMap<SearchNode, SearchNode>();

    my_pq.add(startNode);

    while(!my_pq.isEmpty()){
      updateMemory(my_pq.size() + parent.size());
      incrementNodeCount();
      SearchNode cur = my_pq.poll();
      //If a same node with higher priority exists, discard the node
      if(visited.containsKey(cur) && visited.get(cur)
        <= cur.getPriority())
        continue;
      else visited.put(cur, cur.getPriority());

      //Goal reached, return the result chain
      if(cur.goalTest())
        return backchain(cur, parent);

      ArrayList<SearchNode> suc = cur.getSuccessors();
      for(SearchNode s : suc){
        if(!visited.containsKey(s) ||
          visited.get(s) > s.getPriority()){
          parent.put(s, cur);
          my_pq.add(s);
        }
      }
    }
```

```
35      return null;
    }
```

Line 4-8: Here I use one priority queue and two hashMap. HashMap *visited* stores the key-value pair node and its priority value, HashMap *parent* stores key-value pair node and its previous node.
Line 17-20: I use the method in the instruction to deal with the situation that the same node with different priority found during the search process. The search will put the node into the priorityQueue again only if the node now has a higher priority than before.
Line 28-29: Same idea, only the unvisited node or the same node but with higher priority will be put into the priorityQueue.
For the other part, it is quite similar to the BFS search.

3.4 Output Demostration

I tried the 7 by 7 maze in the instruction. The following is the statistics result compared to the BFS search result. All the input maze files are in the **Input** folder and result GIF files are in the **Result** folder.

```
BFS:
  Nodes explored during search:  37
  Maximum space usage during search 37
  path length: 13
  cost: 12.0
A*:
  Nodes explored during search:  21
  Maximum space usage during search 32
  path length: 13
  cost: 12.0
```

The result animation GIF file **astartResult.gif** can be seen in the folder **result**. And from the result, ouput we can see that with a rather good heuristic, A* search find a same length of the path with less space and time consuming. Here is some states picture, where S stands for the starting point and G stands for the goal point.

```
.......     .......     .......     .......     .......
.##....     .##....     .##....     .##....     .##....
..##...     ..##...     ..##...     ..##...     ..##...
.----..     .----..     .SSS-..     .SSSS..     .SSSS..
.-##--.     .S##--.     .S##--.     .S##SS.     .S##SS.
#-###--     #S###--     #S###--     #S###--     #S###SS
S-..##G     SS..##G     SS..##G     SS..##G     SS..##S
```

4 MultiRobot Problem

4.1 Discussion Problems

4.1.1 State Definition

In the multi robots problem, we need to have the position state for every robot. The numbers we need is 2k - k for x coordinate and k for y coordinate. Furthermore, in this assignment, we introduce the **turn** to reduce the successor's states from a state, prevent redundant states. Represent it as the matrix, it looks as follow:

$$\begin{pmatrix} x_0 & y_0 \\ x_1 & y_1 \\ \vdots & \vdots \\ x_{k-1} & y_{k-1} \end{pmatrix}$$

where there are k robots and each row represents a robot's coordination.

4.1.2 Upper Bound

In the n*n size of maze and k robots, the upper bound of this problem is obviously n^{2k}, as each position can have k robots, including the illegal states.

4.1.3 Collision States

If the number of walls is w, then the number of collision states would be $(n^2 - w)^k - C^k_{n^2-w}$, the total available position minus legal states.

4.1.4 Sparse Maze

If the maze is sparse and the there are a lot of robots, the BFS is not computationally feasible. In an n*n maze with k robot, then each step will generate $a * k^4$, where a is the states from the previous step. It is easy to see that with the exponential increasing it is impossible to solve this problem by using BFS method.

4.1.5 Heuristic Proof

In my implementation, I use the sum of all robots' Manhattan distance as the heuristic, $h = \sum_{i=0}^{k-1} h_i$, where h_i represents the Manhattan distance from robot i to the goal. A function is monotonic as long as:

$$h(N) \leq c(N, P) + h(P)$$
$$h(G) = 0$$

where h is the consistent heuristic function, N is any node in the graph, P is any descendant of N, G is any goal node, c(N,P) is the cost of reaching node P from N.
And this is fairly easy to prove. Consider the single robot situation. In an empty maze, we have $h(N) = c(N, P) + h(P)$. And in a maze with obstacles, the actual cost may larger than h. And append this to the multirobots situation, this heuristic is still monotonic.

4.1.6 Some Interesting Examples

I will discuss this part in detail in the result demonstration part.

4.1.7 8-Puzzles and Heuristic

The 8 -puzzle problem is a special case of multi-robots problem where there is no walls and only one free space that allows the robots to move. The goal is to let each robot be in their corresponding positions.

4.1.8 8-Puzzles Disjoint Set

The 2 disjoint set means that any state in one of the sets is reachable from any other state in that set, but not from any state in the other set. I am thinking use a BFS to search the path for the 8-puzzle problem. First I will pick an arbitrary state as the beginning and use BFS to generate all the possible states. Then I pick another state that not belongs to the set of states to generate from the previous step. Finally, the sum of these two sets' size should be the total possible size of the 8-puzzle problem.

4.2 Code Implementation

4.2.1 getSuccessors

```
public ArrayList<SearchNode> getSuccessors(){
      ArrayList<SearchNode> suc = new ArrayList<SearchNode>();
      Integer[] xNew = new Integer[rob_num], yNew = new Integer[rob_num];
      for(int[] move : moves){
        for (int r = 0; r < rob_num; r++) {
          xNew[r] = robots[r][0];
          yNew[r] = robots[r][1];
        }
        xNew[turn] = robots[turn][0] + move[0];
        yNew[turn] = robots[turn][1] + move[1];

        if(maze_grid.isSafe(xNew[turn], yNew[turn]) &&
          noCollision(xNew, yNew)){

          SearchNode cur = new MultiMazeNode(xNew, yNew,
            getCost() + 1.0, (turn + 1) \% rob_num);
          suc.add(cur);
        }
      }

      return suc;
    }
```

The difference between multi-robot and single-robot is that in multi-robots, **getSuccessors** need to try every direction plus not move action for the robot in that turn.

4.2.2 noCollision

Also apart from **isSafe** method, we need another method **noCollision** to ensure that different robots will not interfere each other. The code is as follows:

```
private boolean noCollision(Integer[] xNew, Integer[] yNew){
      HashSet<Integer> x_cor = new HashSet<Integer>();
      HashSet<Integer> y_cor = new HashSet<Integer>();
      for(int i = 0; i < rob_num; i++){
        x_cor.add(xNew[i]);
        y_cor.add(yNew[i]);
      }
      if(x_cor.size() != rob_num && y_cor.size() != rob_num) return false;
      return true;
    }
```

The idea of **noCollision** is simple, just put each robot's current state into a HashMap and if finally the HashSet has the same number of value as the robots' number, then there is no collision.

4.2.3 getHeuristic

In this part, I use the sum of all robots' Manhattan distance as the heuristic in the multi-robots problem.

```
public double getHeuristic() {
      // TODO Auto-generated method stub
      double res = 0;
      for(int i = 0; i < rob_num; i++)
        res += Math.abs(gx[i] - robots[i][0]) +
             Math.abs(gy[i] - robots[i][1]);
      return res;
    }
```

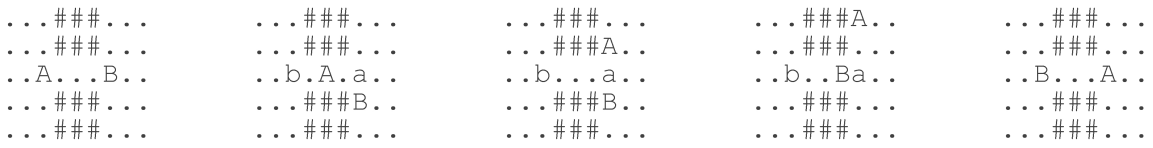
4.3 Result Demostration

In this part, I will show five examples that I think are interesting. All the input maze files are in the **Input** folder and result GIF files are in the **Result** folder.

4.3.1 Yield Example

In this example, one of the robots must yield another robot first so that it can get to its goal. Here show the start and interval states. **A** and **B** are two robots and **a** and **b** are their goals respectively.

```
Multi-A*:
  path length: 19
  Nodes explored during search:  1002
  Maximum space usage during search 1457
```



4.3.2 Reorder Robots

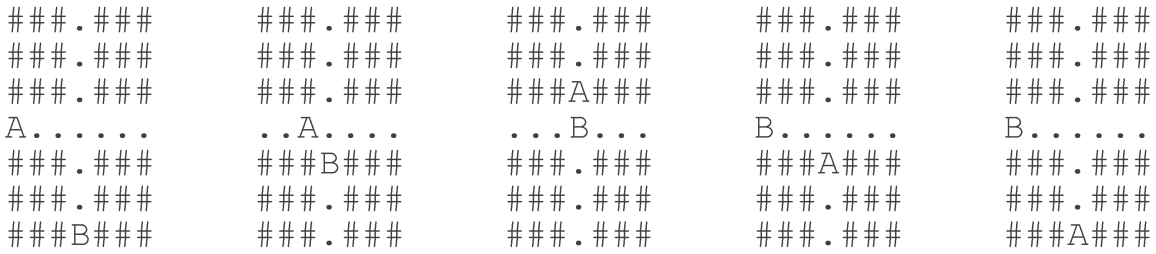
In this example, the order of robots reverses in the goal state. However, the maze is narrow. The uppercase letters represent different robots and the lowercase letters represent their corresponding different goals.

```
Multi-A*:
  path length: 43
  Nodes explored during search:  1154
  Maximum space usage during search 959
```

4.3.3 Crossroad

In this example, two robots are facing a crossroad situation.

```
Multi-A*:
  path length: 20
  Nodes explored during search:  116
  Maximum space usage during search 224
```



4.3.4 8-Puzzles

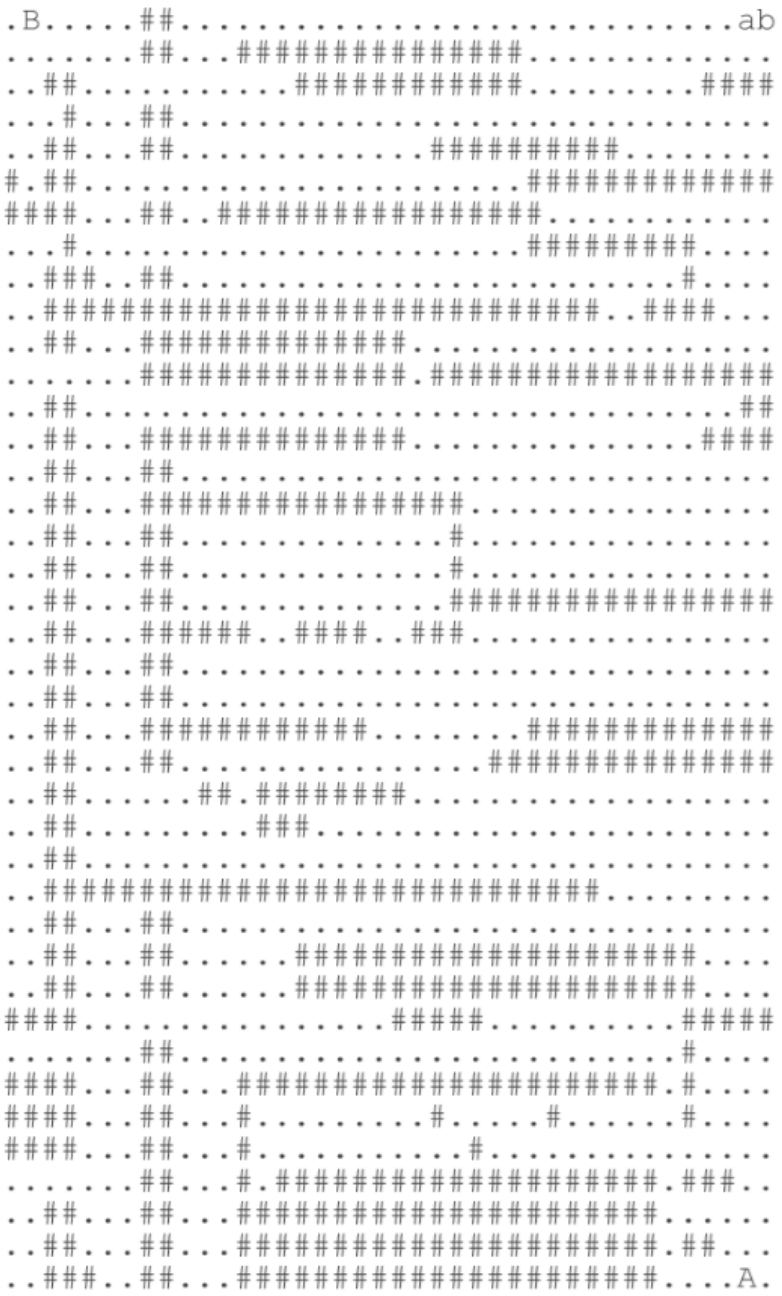
A relatively small maze with a large amount of robots.

```
HDC  HDC  D.C  DCE  DCE
GFE  .GE  HGE  HG.  HGA
BA.  BFA  BFA  BFA  BF.

DCE  DCE  DC.  D.C  .AC
BHA  BA.  BAE  BAE  DBE
.GF  GHF  GHF  GHF  GHF

ABC
DEF
GH.
```

4.3.5 Large Maze

Two robots search for a path in a very large maze.

```
Multi-A*:
  path length: 362
  Nodes explored during search:  2962564
  Maximum space usage during search 1366708
```



5 Blind Robot Problem

5.1 Problem Definition and States

The basic idea of this problem is to converge the belief states into one goal states. The initial states are much like a multi-robot problem, that we assume each floor has a robot. Then we give the robots instruction simultaneously and finally with the merging of states, we have just one left, which is the goal. And our code needs to find this series of instruction to ensure that these states will finally converge to the goal state.

5.2 Heuristic Function

For the heuristic function in blind robot problem, I use the **Manhattan distance** from center position to the goal and the **standard deviation** as the estimation of a current state.

$$\mathbf{c}_x = 1/k * \sum_{i=0}^{k-1} x_i$$

$$\mathbf{c}_y = 1/k * \sum_{i=0}^{k-1} y_i$$

$$\mathbf{SD} = 1/k * \sum_{i=0}^{k-1} Distance((x_i, y_i), (c_x, c_y))$$

where $\mathbf{c}$ is the center point. Then

$$h = \mathbf{Max}(Manhattan(\mathbf{c}_{x,y}, \mathbf{g}_{x,y}), SD)$$

To prove the consistency of the heuristic function, it is similar to the multi-robot one. It is obvious that $h(G) = 0$, as at this point, the center would be the same as the goal, which makes the Manhattan distance to be 0, and the standard deviation is 0. As for $h(N) \leq c(N, P) + h(P)$, we first consider the Manhattan distance part. We can regard it as a single robot problem. The single robot at the center $\mathbf{c}$ is the representative. In an empty maze, we have $h(N) = c(N, P) + h(P)$, then as we give instructions to all the belief states, the center robot $\mathbf{c}$ may or may not change, but the Manhattan distance is still the lower bound it need. And in a maze with obstacles, the actually cost may larger than h, which makes this inequality satisfy. Then we consider the standard deviation part. In the process, the states may diverge or converge. In the diverging situation, the inequality is obviously hold. In the converging situation, the cost would be the sum of distance of each states' change, which is greater than the $h(P)$'s decreasing speed, which means the inequality holds.

5.3 Code Implementation

5.3.1 getSuccessors

```
public ArrayList<SearchNode> getSuccessors() {
  // TODO Auto-generated method stub
  ArrayList<SearchNode> suc = new ArrayList<SearchNode>();
  for(int[] move : moves){
    HashSet<Coor> nextStates = new HashSet<Coor>();
    for(Coor state : states){
      Coor next = new Coor(state.x + move[0], state.y + move[1]);
      if(maze_grid.isSafe(next.x, next.y))
        nextStates.add(next);
      else
        nextStates.add(state);
    }
    suc.add(new BlindMazeNode(nextStates, getCost() +
        1.0 * nextStates.size()));
  }
  return suc;
}
```

The **getSuccessors** method is quite straightforward. Just need to try each direction from the current state and reserve the states that are safe.

5.3.2 Constructor

In **constructor** we need to add every possible position into the *startNdoe*.

```
public BlindMazeProblem(MazeWorld m, int gx, int gy){
  maze_grid = m;
  goal = new Coor(gx, gy);
  HashSet<Coor> startCoor = new HashSet<Coor>();
  for(int i = 0; i < maze_grid.width; i++){
    for(int j = 0; j < maze_grid.height; j++){
      if(maze_grid.isSafe(i, j))
        startCoor.add(new Coor(i, j));
    }
  }
  startNode = new BlindMazeNode(startCoor, 0);
}
```

5.3.3 getHeuristic

As I mentioned before, I use the Manhattan distance from center of all the position to the goal as well as the standard deviation as the heuristic.

```
public double getHeuristic() {
  // TODO Auto-generated method stub
  double centerX = 0.0, centerY = 0.0;
  //Center Point
  for(Coor state : states){
```

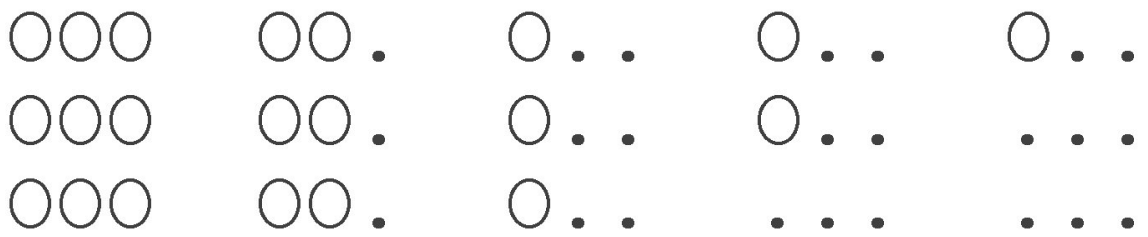
```
6       centerX += state.x;
        centerY += state.y;
8     }
      centerX /= states.size();
10    centerY /= states.size();

12    //Manhattan Distance
      double dist = Math.abs(centerX - goal.x) + Math.abs(centerY - goal.y);
14
      //Standard Deviation
16    double sDiviation = 0.0;
      for(Coor state : states){
18      sDiviation += Math.pow(state.x - centerX, 2) + Math.pow(state.y -
centerY, 2);
      }
20    sDiviation = Math.pow(sDiviation / states.size(), 0.5);

22    return Math.max(dist, sDiviation);
}
```

5.4 Result Demonstration

All the input maze files are in the **Input** folder and result GIF files are in the **Result** folder.

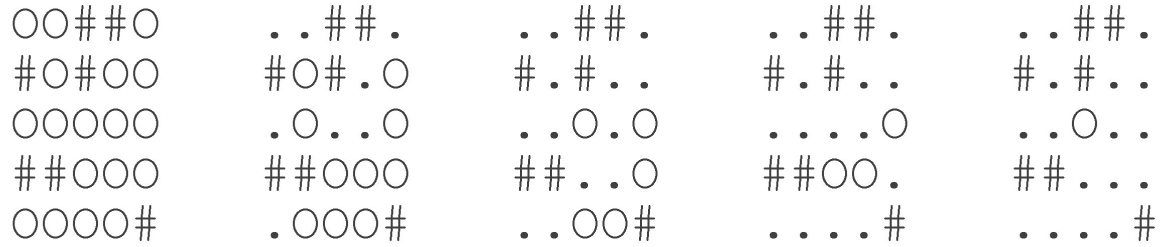
5.4.1 Empty 3 by 3 Maze

```
Blind-A*:
  path length: 5
  Nodes explored during search:  14
  Maximum space usage during search 92
```



5.4.2 5 by 5 maze

```
Blind-A*:
  path length: 11
  Nodes explored during search:  940
  Maximum space usage during search 6574
```



5.4.3 7 by 7 maze

```
Blind-A*:
  path length: 21
  Nodes explored during search:  631969
  Maximum space usage during search 4423777
```

```
OOO##OO     ..O##.O     ...##..     ...##..     ...##..
##OOOOO     ##..OOO     ##O...O     ##.....     ##O....
#OO##OO     #.O##.O     #.O##OO     #.O##..     #..##O.
#O#OOOO     #O#..OO     #.#...O     #.#..OO     #.#O.O.
OOOO##O     ..OO##O     .O.O##O     ...O##O     ....##.
#O#####     #O#####     #.#####     #.#####     #O#####
#OOO###     #..O###     #O.O###     #O.O###     #.O.###

...##..     ...##..     ...##..     ...##..     ...##..
##.....     ##.....     ##.....     ##.....     ##.....
#O.##..     #O.##..     #O.##..     #..##..     #..##..
#.#.O..     #O#O...     #O#O...     #.#....     #.#....
..O.##.     ....##.     ....##.     .O.O##.     ..O.##.
#.#####     #O#####     #.#####     #.#####     #.#####
#O..###     #...###     #...###     #...###     #...###
```

5.5 Discussion: Plynomial-time blind robot planning

It is fairly obvious that the plan always exists if the size of the maze is finite and the goal is in the same connected component of the maze as the start. We can just run a single maze search from each possible start in the maze, and combine all the possible paths together, there has to be one with the same series of instructions as the plan generated by the problem 3's implementation.
As for the motion planner, the main idea is to reduce as many states in each step. First, consider a N*N empty maze. Suppose the goal is at the bottom-right corner, the best strategy would be move right N times, reduce belief states from N^2 to N, the total movement would be N^2 + (N^2-1) + ... + (N + 1). Then we move bottom N times, reduce belief states from N to 1, the total movement is The total movement would be N + (N - 1) + ... + 1. Combine them together the total movement would be O(N^4). So append this to the maze with walls, the strategy would be clear. As we know the maze beforehand, in each step, we just need to find a direction that makes the most robots that cannot move, which means to reduce as much belief states as possible. It is obvious that in a maze with obstacles, in each step the reduced states will be larger than in an empty maze, which means that we can solve the blind robot problem in a polynomial-time if we know the maze beforehand.

6 Literature Review

I am a graduate student.
I read the article Finding optimal solutions to cooperative pathfinding problem. In a multi-robots problem where there are k robots, the state space is N^k. This article tries to implement an efficient method to find the optimal solution in this exponential states space. The basic idea of their method is to do an A* search for each individual robot. Then they tried to resolve the conflict that occurs in the k paths. If the conflict cannot be resolved, they merge those robots into a group with m robots and do an m-robots problem.Furthermore, they even combine the method of iterative deepening to face the real-time demand. The result shows a significant improvement in different types of maze problem. I think this is quite similar to the idea of the quick sort, that in real world multi-robot problem, there may be not that much conflict among these robots and we can save time with this assumption, though in the worst case the time complexity of the method in the article is still exponential.