Chapter 9 Practice for State Space Search

Yonghui Wu

School of Computer Science, Fudan University

yhwu@fudan.edu.cn

WeChat: 13817360465

Chapter 9 Practice for State Space Search

- 9.1 Constructing a State Space Tree
- 9.2 Optimizing State Space Search
- 9.3 A Game Tree Used to Solve a Game Problem

- Search technologies are fundamental technologies in computer science and technology.

- In data structure, search spaces are static, and a search algorithm is to find items with specified properties among a collection of items.
- Three kinds of static search methods for data structure:
  - Sequential Search,
  - Binary Search,
  - Binary Search Trees (BST).

- Sometimes search spaces are dynamic. Searched objects (also called states) are generated during the search.

state space search

- A search problem can be represented as a state space.
- A state space can be represented as an implicit tree or an implicit graph. States are represented as vertices. There are operations that lead from one state to other states.
- The goal for a search problem is to find a path from an initial state to a set of goal states.
- During the search a search tree or a search graph is generated in the state space, called a search space. That is, a search space is a part of a state space.
- DFS and BFS are most widely used dynamic search algorithms.

A search problem can be analyzed from different viewpoints

- **The state space.**
  - The sate space is limited or unlimited?
  - The sate space is static or generated dynamically?
    - For example, in AI, searched objects (states) are generated during the search.

- **The search goal**
  - The search goal is clear or unclear in the state space?
    - For example, in a game of chess, the search goal is unclear.
  - The problem requires you to calculate the goal and / or the paths to the goal?

- **Search**
  - There are any constraints or not for the search?
    - in eight queens' problem, there are constraints among queens, and backtracking is used to find solutions.
  - The search is data-driven or goal-driven?
    - Data-driven search is also called forward search.
    - Goal states are searched from current states. Conversely, the search is goal-driven search, also called backward search.
  - The search is unidirectional search or bidirectional search? If only data-driven search or only goal-driven search is used, the search is unidirectional search. And if data-driven search and goal-driven search are used together, the search is bidirectional search.
  - The search is a game search (that is, there is an opponent) or not? A two-person zero-sum game is a game search, such as Weiqi, Chinese Chess, Chess, and so on.
  - The search is blind search or heuristic search? Heuristic search is using problem-specific knowledge to find solutions.

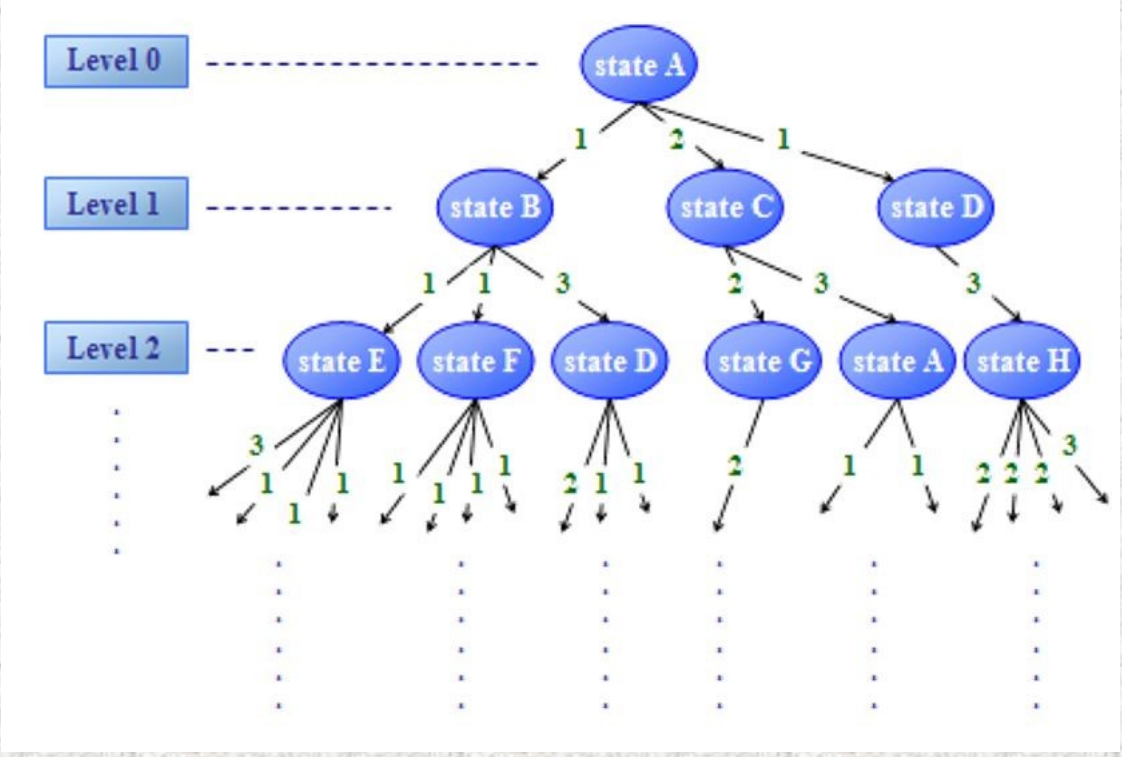
9.1 Constructing a State Space Tree

- A state space consists of a set of states and a set of operations.
  - A state is a situation for a problem.
    - A state can be a situation in a game of chess, or a situation that cars move, stop, or turn on a road, and so on.
  - For a problem, there is one initial state or more than one initial state.

- A state space consists of a set of states and a set of operations.
  - An operation is applied to a state of the problem to get a new state.
    - The relationship between states can be discrete, such as a game of chess;
    - or continuous, such as cars on a road.
    - In a game of a chess, a chesspiece can be moved into another square to change the current state.
    - On the road, cars can move, stop, or turn to change the current state.
  - Operations applied to states can be represented as a successor function.

- If there is only one initial state, the state space is represented as a tree, called a state space tree.
- If there are more than one initial state, the state space is represented as a graph, called a state space graph.

- If there is only one initial state, the state space search is to find a path from an initial state to a set of goal states.
- There are costs for transformations from a state to other states.



- Therefore, to solve a problem of state space search, we need to define states, a successor function, costs, and a state space. For example, in a game of chess,
- States: Chessboards according to rules;
- A successor function: Rules moving a chesspiece;
- Costs: The cost for a state transformation is 1, and represents moving a chesspiece one time;
- A State Space: A set of chessboards according to rules.

- A state space tree is used to represent transformations from an initial state to a set of goal states, and calculate costs for transformations. There are two kinds of cost calculations:
- Evaluating Function $g(x)$: The cost from the initial state to the current state x;
- Heuristic Function $h(x)$: The estimated cost from the current state x to goal states.
- Therefore, if a state space is regarded as a graph, a state space tree can be regarded as a problem for the shortest path.

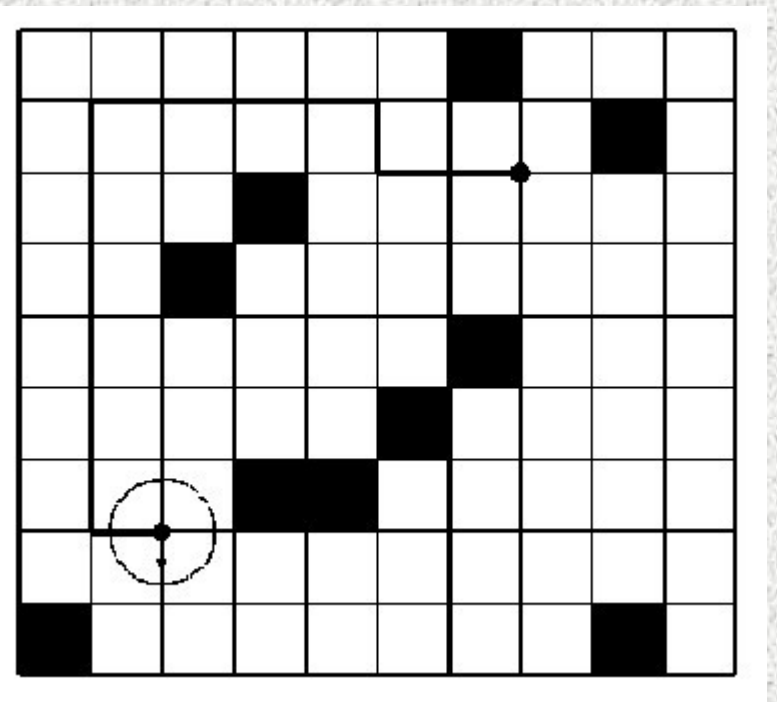
9.1.1 Robot

- **Source: ACM Central Europe 1996**
- **IDs for Online Judges: POJ 1376, ZOJ 1310, UVA 314**

- The Robot Moving Institute is using a robot in their local store to transport different items. Of course the robot should spend only the minimum time necessary when travelling from one place in the store to another. The robot can move only along a straight line (track). All tracks form a rectangular grid. Neighbouring tracks are one meter apart. The store is a rectangle $N \times M$ meters and it is entirely covered by this grid. The distance of the track closest to the side of the store is exactly one meter. The robot has a circular shape with diameter equal to 1.6 meter. The track goes through the center of the robot. The robot always faces north, south, west or east. The tracks are in the south-north and in the west-east directions. The robot can move only in the direction it faces. The direction in which it faces can be changed at each track crossing. Initially the robot stands at a track crossing. The obstacles in the store are formed from pieces occupying 1m $\times$ 1m on the ground. Each obstacle is within a 1 $\times$ 1 square formed by the tracks. The movement of the robot is controlled by two commands. These commands are GO and TURN.

- The GO command has one integer parameter *n* in {1, 2, 3}. After receiving this command the robot moves n meters in the direction it faces.
- The TURN command has one parameter which is either left or right. After receiving this command the robot changes its orientation by 90^{o} in the direction indicated by the parameter.
- The execution of each command lasts one second.
- Help researchers of RMI to write a program which will determine the minimal time in which the robot can move from a given starting point to a given destination.

- The circle is the Robot, black squares are obstacles, and heavy lines are the path that the Robet move through.

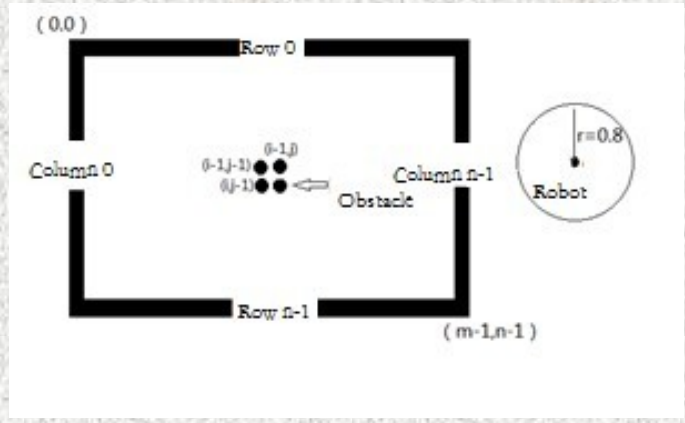


- **Input**
- The input consists of blocks of lines. The first line of each block contains two integers $M \leq 50$ and $N \leq 50$ separated by one space. In each of the next M lines there are N numbers one or zero separated by one space. One represents obstacles and zero represents empty squares. (The tracks are between the squares.) The block is terminated by a line containing four positive integers $B1$ $B2$ $E1$ $E2$ each followed by one space and the word indicating the orientation of the robot at the starting point. $B1$, $B2$ are the coordinates of the square in the north-west corner of which the robot is placed (starting point). $E1$, $E2$ are the coordinates of square to the north-west corner of which the robot should move (destination point). The orientation of the robot when it has reached the destination point is not prescribed. We use (row, column)-type coordinates, i.e. the coordinates of the upper left (the most north-west) square in the store are 0,0 and the lower right (the most south-east) square are M - 1, N - 1. The orientation is given by the words north or west or south or east. The last block contains only one line with $N = 0$ and $M = 0$.

- **Output**
- The output contains one line for each block except the last block in the input. The lines are in the order corresponding to the blocks in the input. The line contains minimal number of seconds in which the robot can reach the destination point from the starting point. If there does not exist any path from the starting point to the destination point the line will contain -1.

Analysis

- First, for a test case, in the area whose coordinate for the upper-left corner is (0, 0), and coordinate for the lower-right corner is (*M*-1, *N*–1), we find grids that the Robot can't move through. The Robot has a circular shape with diameter equal to 1.6 meter, row 0, row *M*-1, column 0, and column *N*–1 are boundaries for the Robot. Therefore, in the area whose coordinate for the upper-left corner is (1, 1), and coordinate for the lower-right corner is (*M*-2, *N*–2), if there is an obstacle at (*i*, *j*), the Robot can't move through (*i*-1, *j*), (*i*. *j*-1), and (*i*-1, *j*-1). That is, (*i*-1, *j*), (*i*. *j*-1), and (*i*-1, *j*-1) should also be set as obstacles.



- **State (*x*, *y*, *s*, *step*):** the current coordinate (*x*, *y*) at which the Robot is, the current orientation *s* that the Robot faces; and the number of commands has been executed *step*.

- **A successor function *move*[ ][ ][ ]:** After the Robot moves *j* meters in direction *i*, the horizontal increment is *move*[*i*][*j*][0] meters, the vertical increment is *move*[*i*][*j*][1] meters, and the orientation is *move*[*i*][*j*][2]. That is, the Robot moves from (*x*, *y*), and moves *j* meters in direction *i*, then the coordinate for the Robot is (*x*+*move*[*i*][*j*][0], *y*+*move*[*i*][*j*][1]), and the orientation that the Robot faces is *move*[*i*][*j*][2]. In order to avoid repeated searches, if the coordinate and the orientation hasn't appeared before, then the command is valid, a new state is generated, and the number of commands for the new state equals to the number of commands for the previous state + 1; else the state is omitted.
- Obviously *move*[ ][ ][ ] are contestant array.
- Byte *move*[4][5][4] = { // the Robot moves *j* meters in direction *i* the horizontal increment is *move*[*i*][*j*][0] meters, the vertical increment is *move*[*i*][*j*][1] meters, and the orientation is *move*[*i*][*j*][2]
- {{0, 0, 1}, {0, 0, 2}, {1, 0, 0}, {2, 0, 0}, {3, 0, 0}},
- {{0, 0, 0}, {0, 0, 3}, {0, 1, 1}, {0, 2, 1}, {0, 3, 1}},
- {{0, 0, 0}, {0, 0, 3} , {0, -1, 2}, {0, -2, 2}, {0, -3, 2}},
- {{0, 0, 1}, {0, 0, 2}, { -1, 0, 3}, { -2, 0, 3}, { -3, 0, 3}},
- };

- **State Space:** a set of state graphs generated by legal commands.
- **Costs:** The cost that the Robot executes one command is 1, represented as an edge in the graph. The number of edges in the path is the number of commands that robot executes from the starting point to the destination point, and is also the minimal number of seconds in which the robot can reach the destination point from the starting point.

- Obviously BFS is suitable to calculate the best path in such a state space. The algorithm is as follows.
- First, the coordinate for the Robot's starting point, the current orientation that the Robot faces; and the number 0 are as the first state. The first state is added into the queue. Then the front for the queue is removed until the queue is empty or the destination point is reached.
- Each time the front is removed from the queue, numbers of meters i ($0 \le i \le 4$) that the Robet moves are enumerated to calculate the reached coordinate (x', y') and the orientation s':
- If (x', y') is an obstacle, then the new state is invalid;
- If (x', y') is the destination point, then minimal number of seconds in which the robot reach the destination point from the starting point is the number of commands for the previous state + 1, and return successfully;
- Otherwise, if (x', y') and s' havn't been visited, the visited mark is set, the number of executed commands *step*'= the number of executed commands for the previous state *step*+1, and the new state containing (x', y'), s' and *step*' is added into the queue.
- The cost for the execution of one command is 1 second. BFS is done layer by layer. If the destination point is reached, the number of executed commands is the minimal number of seconds in which the robot reaches the destination point from the starting point.

9.1.2 The New Villa

- **Source: ACM Southwestern European Regional Contest 1996**
- **IDs for Online Judges: POJ 1137, ZOJ 1301, UVA 321**

- Mr. Black recently bought a villa in the countryside. Only one thing bothers him: although there are light switches in most rooms, the lights they control are often in other rooms than the switches themselves. While his estate agent saw this as a feature, Mr. Black has come to believe that the electricians were a bit absent-minded (to put it mildly) when they connected the switches to the outlets.
- One night, Mr. Black came home late. While standing in the hallway, he noted that the lights in all other rooms were switched off. Unfortunately, Mr. Black was afraid of the dark, so he never dared to enter a room that had its lights out and would never switch off the lights of the room he was in.

- After some thought, Mr. Black was able to use the incorrectly wired light switches to his advantage. He managed to get to his bedroom and to switch off all lights except for the one in the bedroom.
- You are to write a program that, given a description of a villa, determines how to get from the hallway to the bedroom if only the hallway light is initially switched on. You may never enter a dark room, and after the last move, all lights except for the one in the bedroom must be switched off. If there are several paths to the bedroom, you have to find the one which uses the smallest number of steps, where "move from one room to another", "switch on a light" and "switch off a light" each count as one step.

- **Input**
- The input file contains several villa descriptions. Each villa starts with a line containing three integers *r*, *d*, and *s*. *r* is the number of rooms in the villa, which will be at most 10. *d* is the number of doors/connections between the rooms and *s* is the number of light switches in the villa. The rooms are numbered from 1 to *r*; room number 1 is the hallway, room number *r* is the bedroom.
- This line is followed by *d* lines containing two integers *i* and *j* each, specifying that room *i* is connected to room *j* by a door. Then follow s lines containing two integers *k* and *l* each, indicating that there is a light switch in room *k* that controls the light in room *l*.
- A blank line separates the villa description from the next one. The input file ends with a villa having $r = d = s = 0$, which should not be processed.

- **Output**
- For each villa, first output the number of the test case ('Villa #1', 'Villa #2', etc.) in a line of its own.
- If there is a solution to Mr. Black's problem, output the shortest possible sequence of steps that leads him to his bedroom and only leaves the bedroom light switched on. (Output only one shortest sequence if you find more than one.) Adhere to the output format shown in the sample below.
- If there is no solution, output a line containing the statement `The problem cannot be solved.'
- Output a blank line after each test case.

Analysis

- Suppose the interval for the room numbers is [0, r-1].
- **State u:**
- A state u is represented as a r+4-digit binary number,
  - the last 4 digits for u (u%16) represents the current room number,
  - the r-digit prefix for u (u/16) represents the current lights' status for all rooms: one binary digit represents one room's light:
    - 1 represents that the light is on,
    - 0 represents the light is off;
  - for the upper limit for the number of rooms is 10.
- The initial state $u_0=2^4$.
  - the light in the hallway (room 0) is on,
  - lights in other rooms are off.
- The goal state $u_{target}=(1<<(r+4-1))+r-1$.
  - the light in in the bedroom (room r-1) is on,
  - lights in other rooms are off.

- **Successor Function (The rule generating a new state *u_new*):** For state *u*, there are three operations:
  - **Operation 1 — Moving.** If there is a door between the current room (room *u*%16) and room *i* whose light is on, then Mr. Black enters room *i*, and the new state *u_new*=*u*- *u*%16+*i* is generated. That is, room *i* becomes the current room.
  - **Operation 2 — Switching off a light.** If there is a light switch in the current room (room *u*%16) that controls the light in room *i* and the light in room *i* is on (the binary digit corresponding to room *i* in *u*/16 u_{4+i}==1), then a new state is generated *u_new*= *u*-2^{4+i}. That is, the light in room *i* is switched off.
  - **Operation 3 — Switching on a light.** If there is a light switch in room *u*%16 that controls the light in room *i* and the light in room *i* is off (the binary digit corresponding to room *i* in *u*/16 u_{4+i}==0), then a new state is generated *u_new*= *u*+2^{4+i}. That is, the light in room *i* is switched on.

- The generated state *u_new* is valid if the operation meets two following conditions.
  - in order to avoid repeated searches, *u_new* hasn't been visited before;
  - in *u_new*, the light in the current room must be on $(((u\_new/16)\&(2^{u\_new\%16}))==1)$.

- **State Space:**
- From the initial state, new states are generated to construct a state space tree.

- **Cost:**
  - In the state space tree, the cost for each edge is 1.

- The problem requires you to calculate the shortest possible sequence of steps. The upper limit for the number of states is 1024*10. For each state, the upper limit of the number of operations is 30 (10 moving methods + 10 switching on lights + 10 switching off lights). Therefore, BFS is suitable to solve the problem.