Theory of Computer Games, Fall 2018
Homework 1

B04902012 Han-Sheng Liu, CSIE, NTU
E-mail: `b04902012@ntu.edu.tw`

1 Usage

```
./source/bi_bfs < input
```

2 Implementation

2.1 Basic Structure

To store a board, since its walls and goals are constant during a single game, the position of walls and goals are stored globally. In the other hand, the position of player and boxes are stored inside a status of a board.

Precisely speaking, I defined two structures in my code. `coord` and `status`. A `coord` is a structure for storing a coordinate, which consist of x and y, since the sokoban problem is 2-dimensional. A `status` is composed of a `coord player`, and a vector of `coord box`.

To search for a sequence valid movement, it's necessary to memorize whether a status is visited or not. I use a hash table (implemented as `unordered_map` to store this. If two statuses have the same set of boxes, but in different order, they should have the same hash value.

2.2 Pruning

Before searching begins, for every cell, I compute whether it is possible to arrive at least one goal from the cell. This can be done by doing one BFS in the reverse way. After that, all cells which is impossible to arrive any goal are considered *dead*. While searching later, any box should not be pushed into a dead cell. The running time of `large.in` decreased from 109.2 seconds to 67.6 seconds.

Moreover, some pattern of boxes and walls leads to deadlocks. A subset of boxes can no longer be moved when such pattern exists. To check whether a status contains deadlocks or not, I first keep removing boxes that can be moved, until no boxes can be moved. Then, if there are any remaining boxes that does not stand on a goal, a deadlock is detected. The running time of `large.in` decreased from 67.6 seconds to 38.5 seconds.

2.3 Algorithms

- **Breadth-First Search:** A simple bread-first search. The code is `bfs.c`. For `large.in`, it takes 15 seconds to compute the first 9 testcases, but I've never successly computed the last testcase. My computer always crashes, maybe due to memory usage.

- **A*:** A bidirectional A*. The code is `A_star.c`. At the beginning, I did not noticed that the number of boxes can be greater than 5, so I enumerate all permutation of boxes, and calculate the sum of manhatton distance between box_i and $goal_i$. The permutation that leads to minimized sum is chosen. The time complexity is $O(n!)$, which is obviously too big when n grows up to 11. I did not come up with any suitable heurstic function h that never over-estimates, so I took another approach.

  I always sort the boxes and the targets first by x-coordinate then by y-coordinate, and sum over all the manhatton distance between box and target with same order. Although it may over-estimate the cost, the error is bounded. Eventually, I implemented a bidirectional A*, and the running time on `large.in` is only 6 second, but the output is not optimal.

- **Bidirectional BFS:** Apart from searching from the initial status, I also do another BFS which start from all possible final statuses. The number of all possible final statuses is bounded by $4n$, because the player must stand next to box when finished. Once the two searching meets, a solution can be found by backtracking both searching. Note that the branch factors of the two BFS are different. When pushing boxes, the branch factor is 4. However, since we can either pull or not pull when a box can be pulled, the branch factor is 8. To minimize the number of visited statuses, the height of the two searching should be different. After experienment, when height of the reversing BFS around 1/32 of another, the program is relatively faster. The running time on `large.in` is only 14 seconds.

3 Analysis

- **Breadth-First Search:** The complexity should be 4^d.

- **A*:** The complexity should be $d4^d$.

- **Bidirectional BFS:** The complexity should be $4^{d_1} + 8^{d_2}$, where d_1 and d_2 are the height of the two BFS respectively. $d_1 + d_2 = d$.