TCG2023 HW1

b09902004 資工四 郭懷元

How to Compile

Use this command to compile the project:

```
make
```

The source codes will be compiled to a binary called `solve`. `solve` takes input from stdin and writes the solution to stdout. So one can use it this way:

```
./solve < testcases/11.in > solution_11.out
```

Algorithm, Heuristic, Tricks

DFID

Because I only expected DFID to finish the easiest test cases under 5 seconds, I didn't put efforts into move ordering.

I followed the pseudo code in the slides to implement DFID, with the only difference being using recursion instead of maintaining a stack myself.

A*

I followed the pseudo code in textbook to implement A* algorithm.

I made a small change by using heap and hash table to achieve the effect of modifiable priority queue. I pushed all possible next state into the heap first. And when popping a game state from the heap, I will skip that state if its hash has been seen before.

Game state hash

I hash any game state into a 64-bit number. Bit $7(i-1)$ to $7i$
is used to for i-th piece's position (0 means not on board), and other bits are zeroed.

I also tested adding the current index in dice sequence to the hash. This is because there are cases where "looping" is necessary, and board-only hash will not find any solution is such cases. Therefore I use the hash **without** dice sequence index first, and only use the "correct" hash if no solutions were found.

A* heuristic

I only came up with one heuristic. It is as follows:

1. If the goal piece is specified, the heuristic h is the **chessboard distance** ($\max\left(|x_1 - x_2|, |y_1 - y_2|\right)$) between the goal piece and the goal square.
2. If there are no specified goal piece, the heuristic h is the minimum chessboard distance between any piece and the goal square.

This heuristic is trivially admissable (and also consistent), therefore would yield optimal path.

Pruning

During a state in a shortest-path solution, any piece should have one of the following "temporary goal":

1. Waiting to be taken by others
2. On the road to taking another piece
3. Heading towards the goal square
4. No purpose, just wandering around
    - They can also just get closer to the goal square

Thus any move should be "meaningful", where meaningful means the moved piece should either:

1. Move closer to the goal square
    - For goal 3 and 4
2. Move closer or stay exactly 1 step to one of the other pieces
    - For goal 1 and 2
    - Taking a piece is seen as reducing the distance to 0
    - "Staying exactly 1 step" is because we might want to be taken by that piece instead of taking it

By this property, we can prune useless branches by not pushing those states into our priority queue. Theoretically this should at least make heap operations faster by reducing number of items. I also pruned states where the goal piece is eaten, which is impossible to finish.

Other implementation tricks

I extended the `ewn.cpp` and `ewn.h` provided for game-related utilities. Through profiling I found that a significant ratio of time was spent on copying game states, which contains roughly unnecessary 100 integers for the 9*9 board and some shared variable. Therefore, I slimed down the game state class to see if there are any difference.

Experiment Results

Here are some benchmark results. All are compiled with `c++11` and `O3` optimization. Time limit is 10 seconds.

Method	`11.in`	`14.in`	`22.in`	`23.in`	`31.in`	`33.in`	`extra.in`
DFID	0.007s	3.489s	DNF	DNF	DNF	DNF	N/A
A*	0.012s	0.145s	0.839s	1.749s	3.508s	1.284s	N/A
A* + pruning	0.011s	0.066s	0.519s	0.850s	1.857s	0.805s	N/A
A* + slim	0.007s	0.075s	0.370s	0.616s	1.592s	0.665s	4.221s
A* + slim + pruning	0.009s	0.039s	0.278s	0.394s	0.889s	0.441s	3.004s

As the table above shows, pruning can save about $30 \sim 50\%$ of execution time, and slimmed game state can save about $50 \sim 60\%$. Combining both method can reduce the total execution time to less than $\frac{1}{3}$ of the original one.

I also used profiling to check how many states are explored (based on calls to certain function). For test case `31.in`, the algorithm explored around 1.5M states, and it only explored around 0.95M states with pruning.

Inadmissible heuristic experiments

I implemented a inadmissible variation of my original heuristic, where the chessboard distance is replaced by **taxicab distance** ($|x_1 - x_2| + |y_1 - y_2|$). Here is the benchmark result (with slim and pruning)

Distance Type	`11.in`	`14.in`	`22.in`	`23.in`	`31.in`	`33.in`	`extra.in`
chessboard	0.009s	0.039s	0.278s	0.394s	0.889s	0.441s	3.004s
taxicab	0.010s	0.014s	0.024s	0.228s	0.534s	0.136s	1.540s

Because it still takes 1.5s when the admissible heuristic takes 3.0s, I think the cost of getting a sub-optimal is too large and not worth it.

Additional test cases

```
extra.in
--------
9 9
   1   0   0   0   0   0   3   0   0
   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   5   0   0   6
   0   0   0   0   0   0   0   0   0
   2   0   0   0   0   0   0   0   0
   0   0   0   0   0   4   0   0   0

3
4 5 6
1
```

References

People I discussed with: 陳可邦、陳柏諺、王均倍