RushHour Game Solver

Achille Nazaret - Jeremie Klinger

23 janvier 2018

Please have a look at the README.md to have more details about class structures.

1 Solver overview

The solver consists of 5 classes :

1. RushHourGame : The main class implementing the board
2. RushHourGameSolver : Solver class which manipulate RushHourGame objects
3. RushHourGraphic : Fancy graphics class
4. Movement : Class describing a unique movement (car id and number of cells)
5. Vehicle : Class describing each vehicle (id, orientation, coordinates)

In bonus, a generator class is provided to generate custom games.

2 Brute Force Solution

2.1 Overall description

We are using a BFS algorithm to explore all reachable states from the first state S_i, until we come to the final state S_f which we identify by the position of the red car.

Neighbors for state S are defined by all states reachable moving only one car (by eventually several cells). If we store these moves, we are able to backtrack easily from final state to initial state and obtain the shortest path.

2.2 Data structures

— *Visited states* : they are stored as keys of a Hashmap which maps the state's haschode to the movement that led to the state. Doing so we have access to the visited states hashes looking up at the hashmap keys.

— *Backtracking* : In bonus we can reconstruct the sequence of moves that led to the final state by looking sequentially to the hashmap from the final state S_f and applying the opposite movements.

— *State Hash* : Once we know on which row/column lie each car, only its position on the column/row matters. Thus, each state can be identify by a unique String : Car0Position#Car1Position#.... This string is then hashed using java.String.Hash function.

— *Reached states* : The states we have reached and that are waiting to be processed are stored in a *LinkedList* used as a **queue**.

2.3 Brute Force Algorithm

Algorithm 1 BFS algorithm

INPUT : initial state S_i
OUTPUT : List of moves from S_i to S_f

INITALIZE :
boolean **found**, Hashmap **visited**, LinkedList **Q**
found = false
visited $\leftarrow S_i$
Q.$add(S_i)$

while Q $\neq \varnothing$ and **found** $\neq true$ **do**
  $S \leftarrow$ **Q**.$pop()$
  for $T \in$ neighbors(S) **do**
    if $T \notin$ **visited then**
      if $T = S_f$ **then**
        found $\leftarrow true$
      end if
      visited = **visited** $\cup T$
      Q.$add(T)$
    end if
  end for
end while

Backtrack from S_f to S_i

3 A* algorithm

We now try to enhance the bruteforce algorithm by orienting the BFS with heuristics. The queue is turned into a PriorityQueue storing pairs of (State, distance from S_i) The pseudo-code rewrites as follow :

Algorithm 2 BFS algorithm

INPUT : initial state S_i, Heuristic h
OUTPUT : List of moves from S_i to S_f

INITALIZE :
boolean **found**, Hashmap **visited**, PriorityQueue **Q**
found = false
visited $\leftarrow S_i$
Q.$add(state = (S_i, 0), priority = h(S_i))$

while Q $\neq \varnothing$ and **found** $\neq true$ **do**
  $(S, d) \leftarrow$ **Q**.$pop()$
  for $T \in$ neighbors(S) **do**
    if $T \notin$ **visited then**
      if $T = S_f$ **then**
        found $\leftarrow true$
      end if
      visited = **visited** $\cup T$
      Q.$add((T, d+1), priority = h(T) + d + 1)$
    end if
  end for
end while

Backtrack from S_f to S_i

4 Heuristics

Here we described several heuristics.

4.1 Trivial heuristic

A trivial consistent heuristic is given by the function

$$h = 0$$

Using this heuristic is equivalent to execute the depth-first-search algorithm as we did before.

4.2 Blocking cars heuristic

A better heuristic is to associate to a game state the number of cars between the red car and the exit. The *blockingcars* heuristic is obviously *consistent* : let (s, s') be two distinct states and (b, b') the heuristic value for each state. Then to reach state s' from s, the player must move at least $|b - b'|$ cars.

4.3 Move blocking cars heuristic

This is a more advanced heuristic which anticipate the possibility to push away the blocking cars is only one move. If one blocking car cannot free the exit road in only one move (if another horizontal car blocks it), the heuristic

Not consistent (one move can free two vertical cars).