

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
BANGALORE

Modelling Assignment

2016-CS 703

Author 1:
Aneesha Devulapally
IMT2013013

Author 2:
Rishabh Manoj
IMT2013035

November 18, 2016

Contents

1	Project problem	2
2	Modelling the problem	2
3	Goal and CTL specification	4
4	Results	4
5	Conclusion	5

List of Figures

1	Initial and Final Configuration of the Puzzle	2
2	Computation of fixed point approximation for CTL specification	4
3	CTL Counter Example and Execution Sequence	5

1 Project problem

The problem we worked on was Loyd's puzzle, a puzzle that challenges its players to slide the tiles in a certain (initial) configuration to achieve a predefined (final) configuration. The initial and the final configurations are shown below.

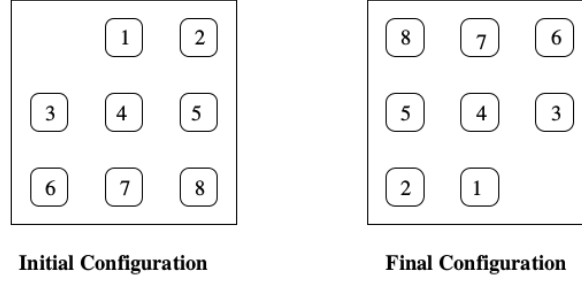


Figure 1: Initial and Final Configuration of the Puzzle

2 Modelling the problem

Loyd's puzzle has an $N \times K$ grid with $N.K - 1$ numbered tiles and a blank tile. There are two arrays, h and v , that keep track of the horizontal and vertical positions of the tiles respectively, such that the position of tile i is given by $h[i]$ and $v[i]$. Tile 0 represents the blank tile. Position $h[i] = 1$ and $v[i] = 1$ is the lowest left corner of the puzzle. Here, we define the action *move* [actions *up*, *down*, *left* and *right*] with reference to Tile 0. For example, if Tile 1 is to the left of Tile 0, the move *left* results in Tile 0 moving *left* and Tile 1 moving *right*. The initialization of all the tiles is done according to the given initial configuration. The next position of each tile is depended on *move* of Tile 0.

Next position of Tile 0 is defined as follows.

```

next(h[0]) := case
  (move = u) & !(h[0] = N) : h[0] + 1;
  (move = d) & !(h[0] = 1) : h[0] - 1;
  TRUE : h[0];
esac;

next(v[0]) := case
  (move = l) & !(v[0] = 1) : v[0] - 1;
  (move = r) & !(v[0] = K) : v[0] + 1;
  TRUE : v[0];
esac;

```

- a. If the action *move* is *up* and Tile 0 is not in the last row, then move *up* ($h[0] + 1$)
- b. If the action *move* is *down* and Tile 0 is not in the first row, then move *down* ($h[0] - 1$)
- c. If the action *move* is *left* and Tile 0 is not in the first column, then move *left* ($v[0] - 1$)
- d. If the action *move* is *right* and Tile 0 is not in the last column, then move *right* ($v[0] + 1$)
- e. Otherwise, remain in the same position.

Next position of Tile i (1-8) is defined as follows.

```

next(h[i]) := case
  (move = d) & !(h[0] = 1) & (v[0]=v[i]) & (h[i] = h[0] - 1) |
  (move = u) & !(h[0] = N) & (v[0]=v[i]) & (h[i] = h[0] + 1) |
    : h[0];
  TRUE : h[i];
  esac;

next(v[i]) := case
  (move = r) & !(v[0] = K) & (h[0]=h[i]) & (v[i] = v[0] + 1) |
  (move = l) & !(v[0] = 1) & (h[0]=h[i]) & (v[i] = v[0] - 1) |
    : v[0];
  TRUE : v[i];
  esac;

```

(Note: The action *move* is with reference to Tile 0)

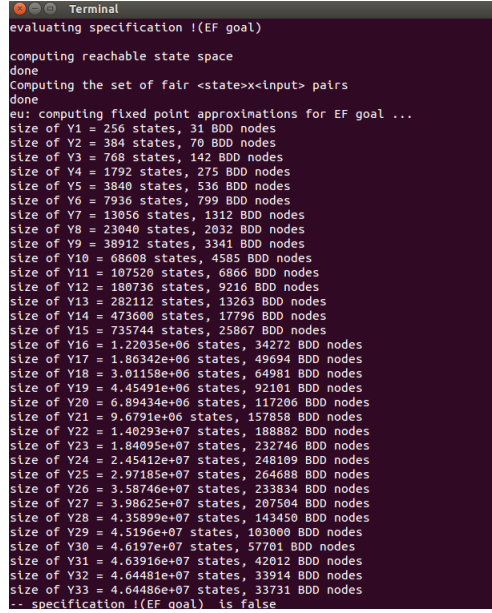
- a. If the action *move* is *down*, Tile 0 is not in the first row, the vertical positions of Tile 0 and Tile i are equal and Tile i is below Tile 0 ($h[i] = h[0] - 1$), **OR**, if the action *move* is *down*, Tile 0 is not in the last row, the vertical positions of Tile 0 and Tile i are equal and Tile i is above Tile 0 ($h[i] = h[0] + 1$), then horizontal position of Tile i is equal to that of Tile 0 .
- b. If the action *move* is *right*, Tile 0 is not in the last column, the horizontal positions of Tile 0 and Tile i are equal and Tile i is to the right of Tile 0 ($v[i] = v[0] + 1$), **OR**, if the action *move* is *left*, Tile 0 is not in the first column, the horizontal positions of Tile 0 and Tile i are equal and Tile i is to the left of Tile 0 ($v[i] = v[0] - 1$), then horizontal position of Tile i is equal to that of Tile 0 .
- c. Otherwise, remain in the same position.

3 Goal and CTL specification

We have defined our *goal* as the conjunction of positions of all the tiles according to the final configuration, and the CTL formula as **!EF(goal)**, which is "Always, globally the defined *goal* is false". When the code is run, it will verify whether the specification is True or False. If the specification is False, it will output a counter example to prove it.

4 Results

The code runs showing the number of BDD nodes while going through all the (10^7) possible states and declares that the specification is false, as shown in Fig:2.



```
Terminal
evaluating specification !(EF goal)
computing reachable state space
done
Computing the set of fair <state>x<input> pairs
done
eu: computing fixed point approximations for EF goal ...
size of Y1 = 256 states, 31 BDD nodes
size of Y2 = 384 states, 70 BDD nodes
size of Y3 = 768 states, 142 BDD nodes
size of Y4 = 1792 states, 275 BDD nodes
size of Y5 = 3840 states, 536 BDD nodes
size of Y6 = 7936 states, 799 BDD nodes
size of Y7 = 13056 states, 1312 BDD nodes
size of Y8 = 23040 states, 2032 BDD nodes
size of Y9 = 38912 states, 3341 BDD nodes
size of Y10 = 68608 states, 4585 BDD nodes
size of Y11 = 107520 states, 6866 BDD nodes
size of Y12 = 180736 states, 9216 BDD nodes
size of Y13 = 282112 states, 13263 BDD nodes
size of Y14 = 473600 states, 17796 BDD nodes
size of Y15 = 735744 states, 25867 BDD nodes
size of Y16 = 1.28035e+06 states, 34272 BDD nodes
size of Y17 = 1.86342e+06 states, 49694 BDD nodes
size of Y18 = 3.01158e+06 states, 64981 BDD nodes
size of Y19 = 4.45491e+06 states, 92101 BDD nodes
size of Y20 = 6.89434e+06 states, 117206 BDD nodes
size of Y21 = 9.6791e+06 states, 157858 BDD nodes
size of Y22 = 1.40293e+07 states, 188882 BDD nodes
size of Y23 = 1.84095e+07 states, 232746 BDD nodes
size of Y24 = 2.45412e+07 states, 248109 BDD nodes
size of Y25 = 2.97185e+07 states, 264688 BDD nodes
size of Y26 = 3.58746e+07 states, 233834 BDD nodes
size of Y27 = 3.98025e+07 states, 207504 BDD nodes
size of Y28 = 4.35899e+07 states, 143450 BDD nodes
size of Y29 = 4.5196e+07 states, 103000 BDD nodes
size of Y30 = 4.6197e+07 states, 57701 BDD nodes
size of Y31 = 4.63916e+07 states, 42012 BDD nodes
size of Y32 = 4.64481e+07 states, 33914 BDD nodes
size of Y33 = 4.64486e+07 states, 33731 BDD nodes
-- specification !(EF goal) is false
```

Figure 2: Computation of **fixed point** approximation for CTL specification

It also gives a path where at the end the defined goal is true (Fig:3a, Fig:3b).

```

Terminal
~$ specification [(if goal) is false
searching Counterexample for & [ TRUE U goal ]
eu_explain: iteration 0: states = 1, BDD nodes = 39
eu_explain: iteration 1: states = 256, BDD nodes = 31
eu_explain: iteration 2: states = 788, BDD nodes = 55
eu_explain: iteration 3: states = 1792, BDD nodes = 100
eu_explain: iteration 4: states = 3840, BDD nodes = 189
eu_explain: iteration 5: states = 7936, BDD nodes = 359
eu_explain: iteration 6: states = 13856, BDD nodes = 597
eu_explain: iteration 7: states = 23040, BDD nodes = 959
eu_explain: iteration 8: states = 38912, BDD nodes = 1474
eu_explain: iteration 9: states = 64080, BDD nodes = 2265
eu_explain: iteration 10: states = 107520, BDD nodes = 3296
eu_explain: iteration 11: states = 180736, BDD nodes = 4532
eu_explain: iteration 12: states = 282112, BDD nodes = 6130
eu_explain: iteration 13: states = 473600, BDD nodes = 8622
eu_explain: iteration 14: states = 757248, BDD nodes = 11467
eu_explain: iteration 15: states = 1.22035e+06, BDD nodes = 16059
eu_explain: iteration 16: states = 1.86242e+06, BDD nodes = 21815
eu_explain: iteration 17: states = 3.01158e+06, BDD nodes = 29835
eu_explain: iteration 18: states = 4.45491e+06, BDD nodes = 38797
eu_explain: iteration 19: states = 6.89434e+06, BDD nodes = 51336
eu_explain: iteration 20: states = 9.6791e+06, BDD nodes = 67624
eu_explain: iteration 21: states = 1.40239e+07, BDD nodes = 86859
eu_explain: iteration 22: states = 1.84095e+07, BDD nodes = 101931
eu_explain: iteration 23: states = 2.45412e+07, BDD nodes = 118403
eu_explain: iteration 24: states = 3.97155e+07, BDD nodes = 151522
eu_explain: iteration 25: states = 5.58746e+07, BDD nodes = 115939
eu_explain: iteration 26: states = 8.98025e+07, BDD nodes = 99088
eu_explain: iteration 27: states = 4.35899e+07, BDD nodes = 76375
eu_explain: iteration 28: states = 4.5196e+07, BDD nodes = 58129
~$
Trace Description: CTL Counterexample
Trace Type: Counterexample
Trace:
-> State: 1:1 <-
  move = u
  h[0] = 3
  h[1] = 3
  h[2] = 3
  h[3] = 2
  h[4] = 2

```

(a) Counter Example

```

Terminal
~$
h[1] = 1
-> State: 1:24 <-
  move = l
  h[0] = 3
  h[3] = 2
-> State: 1:25 <-
  v[0] = 2
  v[6] = 3
-> State: 1:26 <-
  move = d
  v[0] = 1
  v[7] = 2
-> State: 1:27 <-
  h[0] = 5
  h[8] = 2
-> State: 1:28 <-
  move = r
  h[0] = 1
  h[5] = 2
-> State: 1:29 <-
  v[0] = 2
  v[2] = 1
-> State: 1:30 <-
  move = u
  v[0] = 3
  v[3] = 2
  goal = TRUE
Ordering properties by COI size
Properties ordering done
Ordering properties by COI size
Properties ordering done
Ordering properties by COI size
Properties ordering done
Ordering properties by COI size
Properties ordering done
#####
RunTime Statistics
-----
Machine name: perscus-HP-Pavilion-15-Notebook-PC
User time: 40.898 seconds
System time: 9.088 seconds

```

(b) Execution Sequence

Figure 3: CTL Counter Example and Execution Sequence

5 Conclusion

For a 3x3 Loyd's puzzle, we can write a program to model check the problem using a brute force method but for any higher dimension, the number of BDD nodes exponentially increases which makes the brute force method an infeasible option. In such cases, we have to delve into the field of *Artificial Intelligence* which makes use of heuristic techniques as this problem is **NP-complete**.