AI S20 Assignment 1 Report

By Sergey Semushin (BS18-05)

Table of contents

General information

- The code is tested only on SWI-Prolog (threaded, 64 bits, version `8.1.21-198-gd129a7435`).
- The code uses a `clpfd` library for convenient working with integers.
- The heuristic function I used: firstly, try to make steps, then pass the ball diagonally, then pass in a straight line. The motivation for this function was that passing is not needed in most cases, but passing the ball diagonally will help to move it to a further distance.
- Example of using the code:
  - For running random search on `maps/map.pl` map
    `swipl -s main.pl -g main -- --map maps/map.pl --alg random_search`
  - For help
    `swipl -s main.pl -g main -- -h`

Assumptions

- I assumed that more than one orc, human and/or touchdown point can't be at the same coordinates.
- I assumed that a human can pass the ball on a free move.
- I assumed that random search should be runned one time and can make 100 moves maximum.

Structure of the code

On the "lowest level" of my program I defined following fact:
`move_types(ID, FUNCTION, DX, DY, H).`
- `ID` - unique number of move type
- `FUNCTION` - prolog rule which, given `DX`, `DY`, current `X` and current `Y` will return `NEW_X` and `NEW_Y` (or test that a move to these coordinates is possible)
- `DX` and `DY` - for defining direction
- `H` - priority in which moves are made in heuristic search

Then, there is a general `search/6` rule, which implements pure backtracking with no optimization. This rule, as a first argument accepts particular method of search, one of the following:
- `random_search` - defines which move to make at random
- `backtracking_search` - just gives an optimization by cutting branches with too much cost
- `heuristic_search` - changes the order of traversing of the branches and optimizes by the same method as `backtracking_search`

There is also `iterative_deepening_search`, but it is implemented differently

The `main/3` uses `search/6` to find the path, and `search/6` uses `move_types/5` to define which moves are possible is order to split to multiple branches (except `random_search`)

Structure of input files

Each input file should contain:

- Exactly one `size/1` rule
- Exactly one `start/2` rule (In example maps, it is always `start(0, 0)` as the assignment requires)
- Any amount of `h/2`, `o/2` and `t/2` rules.

Proposed test maps

Symbols explanation

- `#` - end of the map
- `S` - starting point
- `H` - human
- `O` - orc
- `T` - touchdown
- `/` - symbol to mark that map can have different sizes

Maps

- **`assignment.pl`**
  Example from the assignment description
  ```
  # # # # # # #
  # . . . . . #
  # . T . . . #
  # H O . . . #
  # . . . . . #
  # S . . . . #
  # # # # # # #
  ```

- **`emptyN.pl`**
  N can be 5, 10 or 20
  This map is supposed to test how good algorithms are on simple maps of different sizes
  ```
  # # # / # # #
  # . . / . T #
  # . . / . . #
  / / / / / / /
  # . . / . . #
  # S . / . . #
  # # # / # # #
  ```

- `human_path.pl`
  To test how algorithms handle free move (and to make iterative deepening search's life easier)

```
# # # # # # #
# . T . . . #
# . H H H . #
# . . . H . #
# H H H H . #
# S . . . . #
# # # # # # #
```

- `labyrinth.pl`
  To test algorithms in "natural" environment

```
# # # # # # # # # # # #
# . . . O . . . O T . #
# . O . . . O . . O . #
# O O O O O O O . O . #
# . . . . . . . . O . #
# . O O O O O O . O . #
# . . O . . . . . . . #
# O . O O O O O O H O #
# . . . . . . . . O . #
# . O O O . O O O O . #
# S O . . . . . . . . #
# # # # # # # # # # # #
```

- `no_touchdown.pl`
  To test algorithms on unsolvable maps and to test time it takes detect that it is impossible to solve

```
# # # # # # # # # # # #
# . . . O . . . O . . #
# . O . . . O . . O . #
# O O O O O O O . O . #
# . . . . . . . . O . #
# . O O O O O O . O . #
# . . O . . . . . . . #
# O . O O O O O O H O #
# . . . . . . . . O . #
# . O O O . O O O O . #
# S O . . . . . . . . #
# # # # # # # # # # # #
```

- `pass_required.pl`
  To test how good algorithms are at passing the ball (and to make heuristic search's life harder)

```
# # # # # # #
# . . . O T #
# . . . O H #
# . . . . O #
# . . . . . #
# S . . . . #
# # # # # # #
```

Part 1. Running tests

In the following table every map-algorithm pair has:

- percentage of failure (when algorithm could not find path when it exists or when algorithm was running for an unreasonable amount of time). If it is not 0%, average path length and time are taken only from successful runs
- average path length (when path is found)
- average time of running (in milliseconds, if not stated otherwise)

Every map-algorithm pair was tested 20 times.

Map \ Algorithm	**random**			**backtracking**			**iterative_deepening**			**heuristic**		
`assignment.pl`	60%	5.0	10.02	0%	17	6.126	0%	3	4.791	0%	17	4.004
`empty5.pl`	25%	53.14	27.03	0%	8	1.738	0%	8	44.84	0%	8	1.320
`empty10.pl`	95%	56	28.0[1]	0%	18	3.283	0%	18	862.6	0%	18	2.151
`empty20.pl`	100%	-	53.22	0%	38	9.201	0%	38	16.9**s**	0%	38	4.071
`human_path.pl`	10%	24.3	24.11	0%	5	0.553	0%	1	7.031	0%	4	0.522
`labyrinth.pl`	100%	-	10.35	0%	26	8.320	0%	18	142.8	0%	25	7.000
`no_touchdown.pl`	0%	-	8.938	0%	-	29.81	0%	-	2.84**s**	0%	-	29.07
`pass_required.pl`	80%	10.0	11.42	0%	11	1.208	0%	3	5.122	0%	19	3.681

By comparing different algorithms, we can conclude several points:

- *Random search* fails to find existing paths too frequently even for small maps.
- *Iterative deepening* method is performing better on small maps. Despite it always returns the best solution, for big maps it runs too long.
- *Backtracking* and *Heuristic* produce very similar results, but heuristic is mostly better. *Heuristic search* was worse than *backtracking* only on specially designed for this map.

Part 2. Bigger field of view

Making field of view 2 instead of 1 can speed up the process of searching for some smart algorithms which remember the map. However, for *backtracking*, *heuristic* and *iterative deepening* and *random* searches it does not change anything, because *random* search performs random moves which does not depend on field of view, and other implemented algorithms just roll back if they failed and have no history.

Doing so can not make an unsolvable map to become solvable, because it does not give any new possible moves. However, it can in theory make search to make less failures by giving insight on where not to pass.

It can not make a solvable map to become unsolvable, because it only increases the abilities of the agent.

[1] If we consider not only one successful run, then average time is 47.134 msec

Part 3. Hard to solve and unsolvable maps

Unsolvable maps

In all unsolvable maps there are either no touchdown points at all (`no_touchdown.pl`) or touchdown points are separated from the initial position by a wall of orcs. Some examples of such maps:

```
# # # # # # #
# . . O H T #
# . . O . . #
# . . . O O #
# . . . . . #
# S . . . . #
# # # # # # #
```

```
# # # # # # #
# . . . . . #
# . . O . . #
# . O T O . #
# . . O . . #
# S . . . . #
# # # # # # #
```

```
# # # # # # #
# O . . . T #
# . O . H . #
# . . O . . #
# . . . O . #
# S . . . O #
# # # # # # #
```

Any other map is solvable, but it might take multiple tries for *random search* or longer time for any algorithm. Algorithms can also produce worse solutions on some maps depending on size, particular arrangement of orcs, touchdown points and humans on the map.

Hard to solve maps

For *random search* and *backtracking search*, the hardness of the map depends only on the amount of moves that do not lead to the goal. So, for *random search* and *backtracking search* `empty20.pl` is harder than `empty5.pl` and even harder than `labyrinth.pl`.

For *iterative deepening search*, map is harder the bigger minimal path is. For example, it runs for a relatively long time on `empty20.pl`, but it's very good on `human_path.pl`.

As for *heuristic search*, the hardest maps are those which differ from the most of maps and therefore they make heuristic function to work against reaching the goal. Particularly, on `pass_required.pl` map heuristic performed worse than *backtracking* and *iterative deepening* algorithms.