Introduction to Artificial Intelligence Assignment 1 - Blood Bowl.

Magomed Magomedov, BS18-05

Core systems

The core game logic consists of the following 3 modules.

Motion

This module provides functionality for 2D point manipulation through 8 basic movements: **up/2, up_right/2, right/2, down_right/2, down/2, down_left/2, left/2, up_left/2**. Furthermore, these private goals are wrapped into 2 public goals: **step4/3** and **step8/3** through the use of *Direction* parameter. The first one is used for human walk, and the second one is for ball pass. Additionally there is a public utility goal **sqr_distance/3** for calculating squared distance between 2 points, use of which we will see when computing cost for *A* Search*.

Game

This module provides rugby gameplay functionality. This is where we store our map data (**map/2, ball/2, human/1**, etc) and this is where map-aware movement happens(**navigate/3, fly/3**). A useful utility in this module is **pass_ball/2** which performs the ball transferring event by placing a human on current position, and removing one from destination position. This effectively means that fact-wise the player currently holding the ball is not present in the knowledge base. Another important feature is **trace_visited/2** goal which, depending on **visited/2** dynamic fact set, calculates the full path from initial ball position to touchdown, returning *Count* integer and *Output* string. Important remark is that, in accordance with the game's rules, a hand off is not counted as a move, but it is printed with the prefix 'H'. Similarly prefix 'P' is used for a passing play. The last feature is **reload_game/0** which reloads map and retracts all **visited/1**.

Map

This file simply supplies a set of facts to be loaded and used by the *Game* module. Available facts are:

- **m/2** - map's top-right corner. Having the bottom-left corner at [0, 0], **m(2, 1)** will result in creating a map 3 by 2, with X coordinate in the range of [0, 2], and Y in [0, 1].
- **b/2** - ball's starting position. Though in the assignment it was specified that we always start from [0, 0], I thought of this as a rather unnecessary limitation and extended system for easier testing. Leaving it as **b(0, 0)** satisfies the assignment condition of starting at the bottom-left corner.

- **h/2, o/2, t/2** - are facts for *human, ork, touchdown* respectively.

Search algorithms

Each search method begins with executing **search/0** goal, which either prints calculated path or “Could not solve”. Combined with file name and **swipl** command, a complete run can be done by executing “swipl -f <filename> -g search”, where <filename> is either *random_search.pl, backtrack_search.pl* or *random_search.pl*.

The following 3 search algorithms were implemented for this assignment, all built upon the core logic described in the previous section.

Random search

Since this method is not guaranteed to find any solution at all, it is being run 1000 times, and for each found solution it is being checked to be with the least *Steps* value. Then the best path is printed. In order to prevent an endless and clearly pointless wandering, the current iteration is stopped when attempting to move into an already visited position. Otherwise, the next point to move at is determined completely randomly by *Direction* in the range [0, 11]. That is **navigate/3** either moves human in one of 4 directions along the 2D axis when *Direction* is in [0, 3], or tosses the ball in one of the 8 directions, adding 4 diagonals to the previous 4, when *Direction* is in [4, 11].

For each point we enter, assert it as **visited/1**. If we are on **touchdown/1** compare the current path with the best stored and then exit. If we are on a regular point, try performing a random of the 12 movements described above, and if we landed onto an unvisited point, call **search/1** on that new point. Of course, if during any of our moves we are stepping on a point with a human already standing here, perform a ball pass by calling **pass_ball/2**.

After all 1000 iterations, if there is a **best/2** - print it, otherwise - “Could not solve”.

Backtrack search

Here we as well assert the current point at **visited/1**, but in order to restore the state of the map after going back, we also retract the point at the end of the iteration. Similarly to the previous technique, upon arriving on **touchdown/1** we trace and try to update our best path with the current one depending on *Steps* integer comparison. Otherwise, for each of the 11 elements we launch a new recursion branch. Important detail here is that the new branch launching sequence ends with **;true** in order to continue with the other branches when one fails. We understand that this is not an authentic way of thinking in Prolog, but it works nevertheless.

After all branches are explored, we either print the best route or the failure message.

A* search

In order to facilitate our core logic to work for this method we needed to implement some additional goals.

try_cost/2 assigns cost of to point, if not assigned yet, based on a simple heuristic: Cost = distance from start + distance to touchdown.

update_pending/2 is used to register new points to be explored after we've made a move. This function is called for every of the 12 directions when we step on a new point. The key feature here is that it also calls **link/2**, i.e. writes from which source point we are going to make a move to this pending one. This is very useful for building the final path because in A* we tend to visit many other points that do not necessarily belong to the optimal path as we explore the map.

link_visited/1 with **link/2** traverses all the way back from **touchdown/2** and restores the optimal path. Additionally, it restores final human placement in order for **trace_visited/2** to be able to determine whether a move is a hand off, a toss pass, or neither of these.

With this A*-specific functionality the core loop becomes simple. Upon arrival on a point, retract it from **pending/1**, assert it to **visited/1**. If **touchdown/1** then restore optimal path by calling **link_visited/1**, then collect points by **trace_visited/2**, and finally print result with subsequent return to the top-level **search/0** goal. Otherwise, update pending cells around this one in the loop, obtain the best available **pending/1** point based on the cost, and run another iteration.

Examples

Here are a few examples of posible map setups and some remarks regarding them. Even though all the maps are run against *A** method, the other 2 algorithms obtain the same result as well. For each case there is *map.pl* content and console output. As a reminder, after putting input into *map.pl* the program is run by "swipl -f astar_search.pl -g search". Finally, it worth saying that being able to see 2 cells away does not improve my implementation in any substantial way.

Corridor

Here we have a simple narrow path [6, 0]. No orks, the ball with humans is at [0, 0], there is another human at [4, 0] and the touchdown is at the opposite end - at [6, 0]. Here we have 2 options: either walk to the second human by 3 steps, hand off the ball, and let the other one finish with 2 more steps,

resulting in 5 rounds total. However, passing the ball by toss to the second human directly from the initial position, and then letting him make his last 2 moves would take only 3 rounds.

map.pl

```
m(6, 0).
b(0, 0).
h(4, 0).
t(6, 0).
```

console

```
3 turns:
P 4 0
  5 0
  6 0
1 msec
```

Barricade

This is a representative example because it forces us to utilize diagonal ball passing mechanic, as there is no other way to resolve this match. So the first human would go 1 cell right, pass the ball over top-right diagonal, and the second human would bring the ball to the touchdown. All that action would take 3 rounds.

map.pl

```
m(2, 2).
b(0, 0).
h(2, 1).
o(0, 2).
o(1, 1).
o(2, 0).
t(2, 2).
```

console

```
3 turns:
  1 0
P 2 1
  2 2
0 msec
```

Queue

This is the map that breaks A*, because A* does not take into account the rules of the bowl, it only thinks about physical distance travelled. And physically speaking, what it does is the right way, because it utilizes diagonal pass, which is shorter than walking axis-aligned lines only. However, since backtracking actually checks all possibilities, unlike A* which stops at the first path, it finds the actual best solution.

map.pl	**console (A*)**	**console (backtrack)**

`m(2, 2).`
`b(0, 0).`
`h(0, 2).`
`h(1, 2).`
`h(1, 0).`
`h(2, 0).`
`h(2, 1).`
`o(1, 1).`
`t(2, 2).`
	`3 turns:`
	`  0 1`
	`P 1 2`
	`  2 2`
	`3 msec`
		`1 turns:`
		`H 1 0`
		`H 2 0`
		`H 2 1`
		`  2 2`
		`16 msec`