Team $-i^2$
AI For Games Coursework Submission (Tanks)

Connor Aspinall George Bell Denis Torgunov

Contents

1 Introduction

This report details the implementation of a subsumption-based AI, as developed by the group "Team $-i^2$".[1]

2 AI Design

The AI produced by our group for this coursework focuses on utilising a subsumption architecture, inspired by the work done by one of the group members in the context of intelligent robotics. The subsumptive architecture, as proposed by Brooks, aims to form close connections between sensory inputs and behaviours of an intelligent agent [1]. In the context of an intelligent Tanks player, we can take information gleamed from the state of the map as simulating sensory inputs, and define a series of behaviours (such as running away, exploring, or seeking an enemy) to take in specific situtations. This approach lends itself well to group work, as individual components and behaviours can be developed separately, and then combined in arbitrairly complex ways using the subsumption architecture.

The binding of game situations (sensory inputs) to behaviours can be changed dynamically, or assigned at the start. For the purposes of this coursework, we define 2 major "behavious profiles" (as discussed in section 2.2), a "Hunter" and an "Explorer". For the discussion of potential for dynamic and adaptive behaviour see section 4.

2.1 Subsumptive Control

In order to make it possible to implement the subsumptive control system, we have created a class, `SubsumptionDispatch`, which calls the corresponding action when the right "stimulus" occurs. For the full source code listing please see section **??**. In this section we will only highlight the central parts of the implementation.

Firstly, we make use of C# delegates to allow us to use higher-order methods, passing methods around as values and evoking them when needed. Specifically, we define the following two delegates:

```
public delegate bool Situation();
public delegate ICommand Action();
```

A `Situation` is a predicate that represents a situation in the game, corresponding to sensory input in a traditional subsumption architecture. An `Action` represents a method that returns the next command to execute, corresponding to operating an acutator. It is assumed to take no arguments, relying purely on the internal state of the main class for any information about map composition, current player position, etc.

Having defined those delegates, we can create a dispatch table in a linked list of tuples:

```
private List<Tuple<Situation, Action>> dispatchTable;
```

We assume that the first element in the list has the highest priority. If its `Situation` arises, we execute the corresponding `Action`. Otherwise, we continue down the list. We assume that at least one of the `Situation`s will occur. In order to ensure that, the final

[1] Refine and add abstract

element of the list should be a "default" action: simulated using a predicate that is always true.

With those definitons, we can now create the main program loop: the `act()` method:

```
public ICommand act()
{
  foreach (Tuple<Situation, Action> behaviour in dispatchTable)
  {
    if (behaviour.Item1())
    {
      return behaviour.Item2();
    }
  }
  return null;
}
```

As soon as the `Action` whose `Situation` predicate returns true is found, we simply execute that action, returning the corresponding command to be executed on this turn.

2.2 Behaviour Profiles

We consider two possible behaviour profiles for is AI: the Hunter and the Explorer. The goal of the game it to maximise the score, and the behaviour process aims to do so in the most efficient way. If exploration is more valuable, point-wise, than killing opponents, we can utilise the Explorer profile, which focuses primarily on discovering the map and avoiding confrontation. On the other hand, if the amount of points received for killing an enemy is higher than the amount of points gained (on average) through exploration, we utilise the Hunter profile, which will actively seek to destroy enemy tanks.

Figure 1 outlines the Explorer behaviour profile, and figure 2 outlines Hunter. The Explorer aims to explore as much of the map as possible, while allowing the enemies to fight amongst themselves, only going on the offensive when there is a single enemy left. On the other hand, the Hunter actively seeks to gain points by destroying enemy tanks, and only then explores the remaining parts of the map.

At present, we adopt the Explorer profile if the average score gain for discovering a rock or empty cell is larger than the score gain for destroying an enemy. This could be further improved, however appears to be a good "ball-park" metric.

3 Testing and Tuning

4 Recommendations for Future Work

As mentioned earlier in section 2, it is possible to make this AI better by, instead of providing predefined behaviour profiles, allowing for a machine learning aspect by allowing the AI to learn, throughout the game, to associate specific `Situation`s with the best `Action`s, as well as adjusting the priority of each binding accordingly. Genetic programming might be a good fit for such a task. However, because it is impossible to save the information on the GridWorld servers themsevles between games, it means that the AI has to learn locally, which severely limits the applicability of this approach within the given setup.

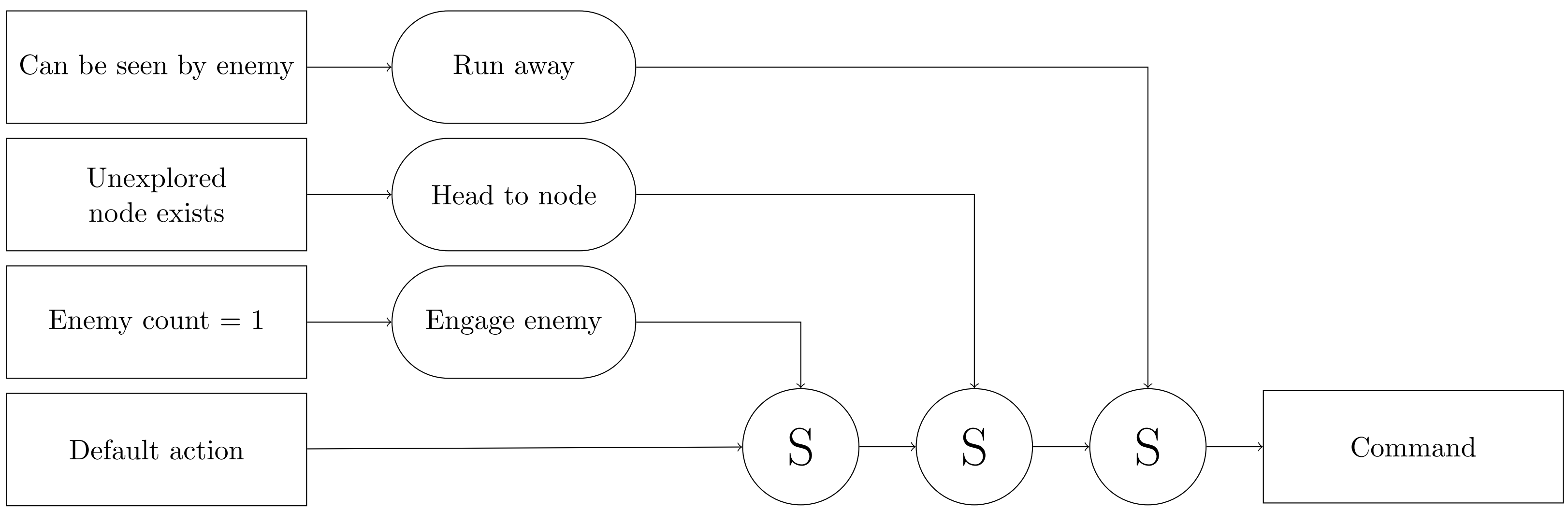

Figure 1: The Explorer profile

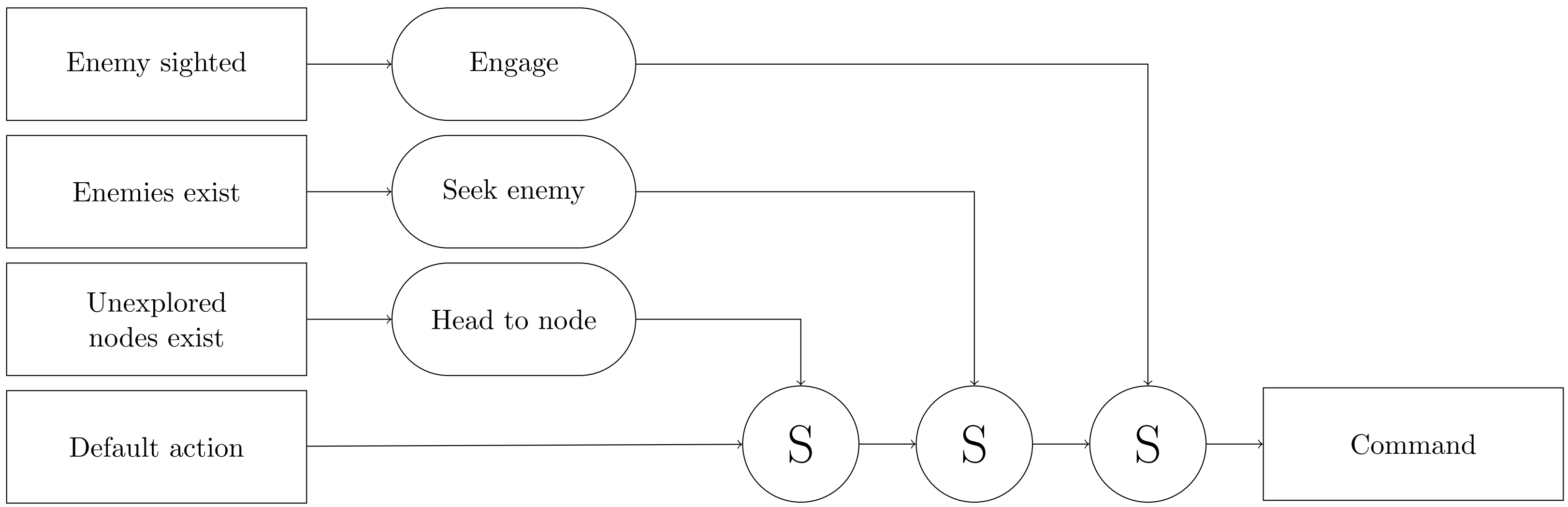

Figure 2: The Hunter profile

The actions and predicates themselves could be further refined and optimised[2].

Our approach to mapping and extracting infromation from the local map kept is, most likely, the least optimised part of the implementation. In particular, we have to loop through the whole array multiple times on each game loop iteration. Combining the tests performed into a single loop and making predicates simple accessors to the boolean fields holding the information gathered this way should make the code a lot more refined.

Finally, the pathfinding algorithm, as presented in the code, computes the path anew on each loop iteration. If the target cell has not changed, and there is no change in `Situation`, it should be possible to cache the path initially computed, and simply continue moving as needed, greatly reducing the work done on each game loop iteration[3].

4.1 GridWorld Recommendations

As mentioned above, the main element that we found missing in both this and the previous coursework submission for this module is the ability to retain information between runs on the GridWorld server. If each user was allocated a small amount of storage on the server, or a way to redirect some form of data to a text or binary file that could then be retrieved, it would be possible to view the server as a dynamic learning environment, broadening the applicability of machine learning approaches.

5 Conclusions

[2] George, please fill out as needed regarding the actions

[3] Connor, please make sure this is right

References

[1] R. A. Brooks, "How to build complete creatures rather than isolated cognitive simulators."

A Evidence of Group Work

Throughout this project's development, Github has been used in order to synchronise our work and communicate issues. Below, we list the final log of git contributions, showing that all the group members have contributed. Note that, on average, people with less commits had more code added per commit.[4]

B Source code

[4] Add git logs when done