Artificial Intelligence for Games
Developing an AI agent for the game of Tanks based on the subsumptive architecture

Connor Aspinall (13021267) George Bell (13025221)
Denis Torgunov (13011546)

Abstract

This report details the design and implementation of an AI player for a game of Tanks, based on a subsumptive architecutre. We highlight the improvements to our pathfinding module based on what we have learned during our work on the previous coursework, as well as detail our own implementation of subsumptive dispatch. We mention some of the issues we had during development and provide suggestions for future work and improvement.

Contents

1 Introduction

This report details the implementation of a subsumption-based AI, as developed by the group "Team $-i^2$". The implemnetation focuses on an adaptive approcah and uses A* for effiecent pathfinding and navigation. It is meant to be extendible and the implementation produced, while succesful[1] during testing, was envisoned more as a proof of concept, with suggestions on better levereging the approach proposed given in section 4.

Due to the fact that most of the codebase had to be implemented from scratch, as well as time constraints, the resulting AI might not be sufficiently optimised, and therefore unable to utilise CPU time provided in the most efficient manner. Suggestions for further optimisation are also given in section 4.

2 AI Design and Implementation

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 arbitrarily 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.

[1] was it?

Having defined those delegates, we can implement a dispatch table as 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 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` with a matching `Situation` 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 this AI: the Hunter and the Explorer. The goal of the game is 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.

The choice of profile depends on what will give us a higher potential score gain. However, since we did not have sufficient data regarding how either profile performs against other AI opponents (see section **??** for details), we instead choose to randomise the choise, making it possible for either approach to be chosen when the AI is first loaded in. But, in order to maximise our chanses of winning, we bias the randomisation based on score gains for a given terrian.

In particular, as mentioned above, Hunter aims to gain points for killing enemy tanks before other players can do so. But we can only destroy 3 tanks in total, and the gain in points might not be worth the risk. On the other hand, Explorer attempts to avoid

confrontation, which might allow the enemy to cut us off from certain portions of the map, diminishing our score.

In the end, we chose to use the following metric: if the average gain for exploring 3/4 of the map is higher than the points gained for killing all 3 tanks, we bias the random profile chooser towards Explorer. Otherwise, the bias is towards Hunter. The "biased" profile has a 60% chance of being picked.

Since the default action implies that there are no enemies left and all of the map has been explored, we simply stay still, as there are no further points to be gained.

2.3 Mapping and Pathfinding

In order to make it easier to navigate and seek out enemies, we keep track not only of the visible area, but also of the overall map explored so far. To that end, we have defined a `Cell` enum. We keep and update an array of `Cell`s, filling it with the "visible" information on each turn. [2]

We also keep track of the last place we saw enemy tanks, to make it easier to avoid or chase them later. This means that we also have to "prune" the map every so often, making sure there is at most one instance of each enemy (the one we saw most recently). For more details, please see section **??**.

The pathfinding algorithm we use is a refined and improved version of the A* implementation used in our previous coursework submission [2]. We have cleaned up the `GridNode` class, making it more streamlined, smaller and easier to use by removing redundant accessor functions. Since the class is only ever used within the project and no pre-processing of parameters is necessary, and no guarded conditions exist that could be enforced by, we simply made the relevant fields public. The only exception is `GridNode.fCost`, which has been replacted with a getter that simply returns the relevant value, computed based on the other parameters. This reduces the risk of `GridNode.fCost` not being updated properly by mistake.

In addition, we also have completely restructured the main pathfinding logic, in order to remove errors previously present. We have levereged what we have learned while implementing the first iteration [2] and improved on the main shortcomings.

The most significant changes are as follows:

- Fixed an issue which caused the `GridNode.fCost` to be set inconsistantly while constructing the list of neighbours.
- Fixed an issue where the path returned is not gaurenteed to be the shortest path as some `GridNode`s where skipped unnecessarily.
- Optimised the implementation by sorting the list to act as a priority queue, potentially reducing CPU time (depending on the native .NET sort implementation).
- To improve code readability, we use a stack when listing neighbours. We loop through the stack's contents instead of considering each neighbour explicitly when testing whether a coordinate is within the bounds of the board array. Previously, a nested loop was

[2] What the word?

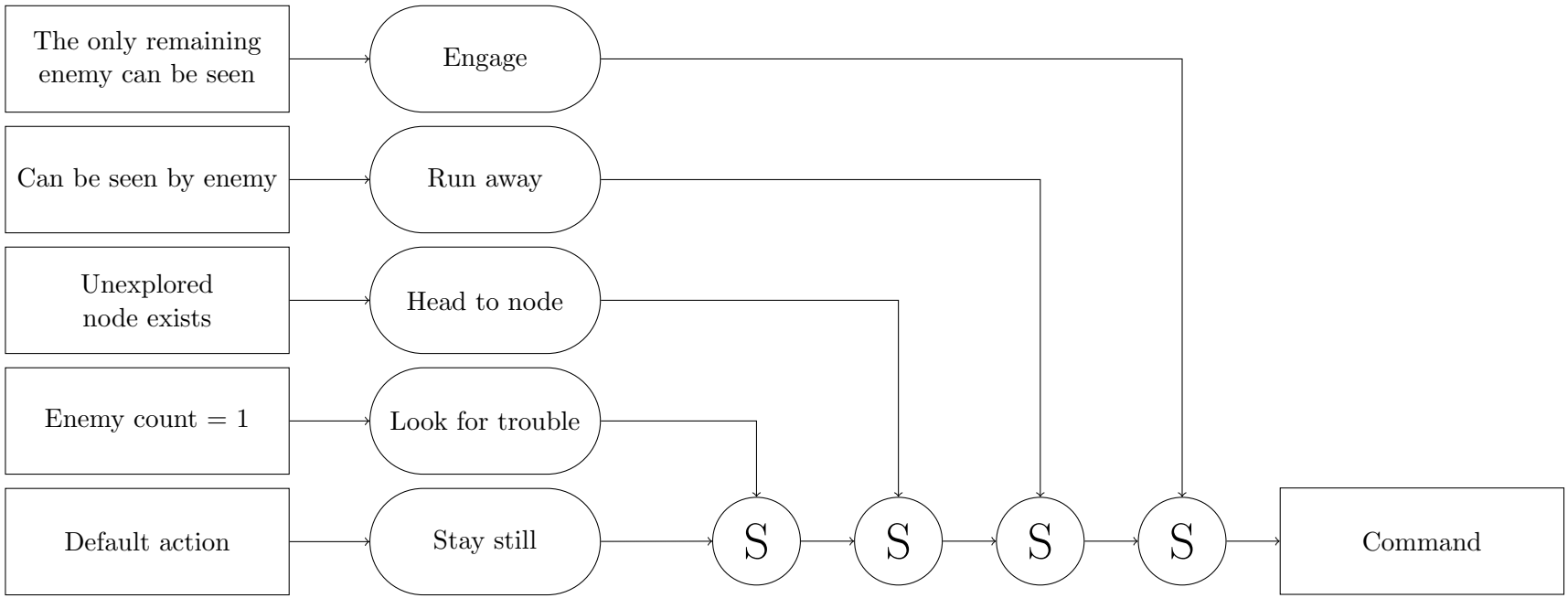

Figure 1: The Explorer profile

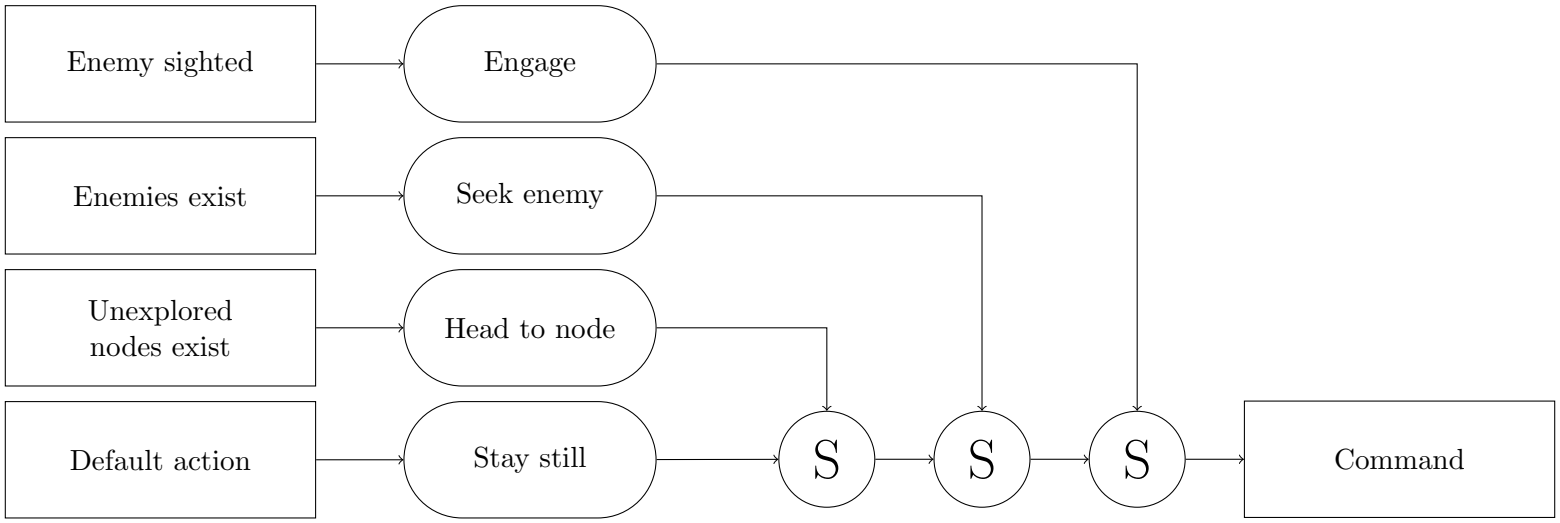

Figure 2: The Hunter profile

used, but the corrent approach, in addition to making the code easier to understand, also reduces complexity.

For the full listing of the pathfinding code please see section **??**.

Furthermore, as several actions in our behaviour profiles involve navigating to unexplored parts of the map, we needed a reliable way to determine which unexplored cells we could navigate to. In order to achive that, we have implemented a function `LocationLocator.UnexploredNode()`, which returns a node at the edge of the currently explored region and not behind a potential wall, if such a node exists. The full code listing is available in section **??**, and is based on [3].

2.4 Actions

The actions that are carried out by the AI are dictated by the Subsumption Dispatch Table, which calls the relevant method in the class `LocationLocator` that, in turn, attempts to find the best location to travel to in the given situation. This is achieved by having several methods, each relevant to one of the potental calls from the Subsumption Dispatch Table, with one being called each turn .

2.4.1 Retreating

One of these key actions that can be carried out under the Explorer profile (as can be seen in Figure 1) is the run away action. This allows the AI to retreat when threated by an opponent, with the corresponding action being found in the method `LocationLocator.Retreat()`. The full source code is listed in section **??**. This method begins by generating a `List` of `GridNode`s found around the player that contain an obstacle that can be hidden behind. This list is initially generated from a 3-by-3 area which is centred on the player, however this will be expanded if no suitable hiding locations are found. Due to the increasing complexity of this algorithm upon expansion [4], it is capped at an area of 5-by-5, or an additional `GridNode` in each direction. If, after analysing an area of that size we find no suitable cover, it will default to dodging in a random direction until it moves closer to something it can take cover behind.

Upon finding a valid piece of cover, the AI will check behind it to see if it is a valid location. This is done by comparing the value of the `Cell` to that of `Cell.Empty`. Initally, this is done by looking at any obstacles in the four cardinal directions within GridWorld (Up, Down, Left and Right) and check the side opposite the player. If the these locations are not valid, or if no obstacles exist in these positions, the AI checks the corner `Cell`s. These can potentionally have two seperate locations where the tank can hide, so both the furtherest X and Y locations (relative to the player) must be checked. Again, if there is a valid location to move to and hide at, it will do so, but if there is not then the AI defaults back to the `LocationLocator.RandomDodge()` method.

`LocationLocator.RandomDodge()`, which is used when no hidding places are found close to the player, works to determine where the player should move. The randomness is included so that, if there are multiple moves possible at a given time, it will choose one that cannot be predicted. This behaviour is often seen in video games [5], as it can lead to unexpected events occuring for the player, although it is also helpful in this scenario as it enables the AI to be slightly unpredictable, potentally dealing with opponents that implemtent machine learning.

2.4.2 Attacking

The key action from the Hunter profile (Figure 2) is to find and kill a visible opponent. This action is carried out in the `LocationLocator.Attack()` method, which accepts the following arguements:

- The `GridSquare` that contains the opponent to attack.
- A `PlayerWorldState.Facing` that represents the direction the AI is facing.
- The `Command` to be executed if the opponent cannot be attacked directly.

To begin with, the method calculates if it is possible to kill the opponent this turn, proceeding to do so if it is possible. This is achieved by checking if the AI and the opponent share either an X or a Y coordinate. If they do, then the AI calculates whether the tank is currently facing the opponent, before either attacking, or rotating and then attacking the given opponent. If this is not the case, then the `Command` is returned, which is part of a path for the AI to follow towards the opponent. As the tank can only move in four directions, and cannot shoot diagonally, this ensures that the tank will reach a point from which it can attack the given enemy.

3 Testing and Tuning

In order to test pathfinding and exploration without the oppontent AI destroying us, we have implemented a simple "Useless" AI, following the logic described in the lecture notes, that simply stays stationary and avoids attacking. This allowed us to safely navigate around the map and discover the bugs in the exploration part of the code.

In addition, problems arouse due to unexpected errors coming from Visual Studio. Firstly, we got errors trying to load the (standard) `Tuple` class, so we have implemented our own, adhereing to the same naming conventions for elements [6], to make it interoperate with the code already written. In addition, similar errors appeared as the development progressed. Eventually, we have decided to try changing the .NET framework version used by the IDE to 3.5, and while that seemed to resolve the issue, it gave us cryptic warnings that we did not have time to investigate in detail and that might account for some of the bugs.

Due to problems accessing the GridWorld server, all of our testing had to be carried out locally, utilising either the "Useless" AI or the stock AI provided. Originally, we were going to tune the performance of the agent by experimenting with different priorities and actions in the behaviour profiles, but as real world performance data was unavailable, it could not be done reliably.

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, introducing 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.

The actions and predicates themselves could be further refined and optimised. As detailed in section 2.4, the complexity of some actions may be increased, allowing the AI to make more informed descions. With additonal refinement and testing, we could find the right balance between runtime and complexity. For attacking the AI carries out, the main addition to make to the predicates would be to calcutate whether the player can attack the opponent before they themselves can be attacked, enabling a form of 'fight or flight' mechanism to be implemented.

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 less redundant.

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.

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.

More clarity in representing the game situation and knowledge would make debugging easier. For example, some form of visulisation of a tank firing, as well as a way to simulate the agent's field of view within GridWorld.

5 Conclusions

While developing this AI, we have levereged the skills and knowledge gained both as part of this module, as well as other related subjects. It has provided us with an interesting challenge, and the final intelligent agent produced has a lot of potential. While the submitted version might not be sufficiently tested and optimised, we believe that it could form a strong foundation for an adaptive, intelligent player with emergent intelligent behaviour.

References

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

[2] C. Aspinall, G. Bell, M. Coyne, C. Cummings, C. Eidson, and D. Torgunov, "Artificial intelligence for games. Developing and AI agent for the game of Snails," March 2016, coursework submission for AI for Games (CM-0328D).

[3] D. Torgunov, "Intelligent robotics. Maze exploration," April 2016, coursework submission for Intelligent Robotics (CY-0330D).

[4] A. Sherrod, *Data structures and algorithms for game developers*. Cengage Learning, 2007.

[5] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 9, no. 1, p. 1, 2013.

[6] Tuple class. Microsoft. Accessed: 04 May 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/system.tuple%28v=vs.110%29.aspx

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 participated. Note that, on average, people with less commits had more code added per commit. The majority of the work on the report has been done as a group.

B Source code