Implementing a high-level planner into a Real-time strategy game using the iThink planning library

Andrew Roper - 100127617
University of Derby Derby, Derbyshire
Email: contact@andrewroper.co.uk

Abstract—In this paper we will be looking at how a high level planner can be implemented into a real-time strategy (RTS) game. Using planners in games is one method which can be used to orchestrate actions for Artificial Intelligence (AI), and allows for different plans to be created with different plots. This paper will also examine how a planner can be incorporated alongside environmental data, such as influence maps and emerging agent knowledge of the statespace.

I. INTRODUCTION

Planning systems have been used in a range of games to manage the actions for the Artificial Intelligence (AI). As the scale of games increases, so do the challenges faced to develop a believable AI system as they come across new challenges [1]. When the first challenges with AI implementation were come across, developers started to use A* pathfinding [] for navigation, whilst also using Finite State Machines(FSM) and Rule Based Systems(RBS) for determining dynamic behaviour [1].

With recent advancements in gameplay design moving more towards open-ended gameplay coupled with an increased scale of development, new options need to be thought of and integrated. Agents need to be able to dynamically react to the world, and learn it’s surroundings to calculate alternate routes to find solutions to problems [1].

Using planning architectures in games has recently become more popular due to the increase in scope, and required adaptability, of artificial intelligence within games. One genre of game that may benefit from using a STRIPS planning architecture is the Real-Time Strategy (RTS) domain, where the AI needs to constantly re-evaluate the world as it learns, generating new plans as it does so.

This paper will be aiming to achieve the following:

• Implementing a STRIPS Planning architecture into an RTS environment
• Determine the feasibility of using planners in RTS games
• Determine if STRIPS planners can learn, and the impact of learning

Furthermore, we will be looking into finding out what are the benefits to using a STRIPS planner, are there any drawbacks, or reasons that a STRIPS planner should be avoided or used?

II. RELATED WORK

In recent years, there has been an increase in the number of games that are using planning systems. This is due to the advancement in AI, alongside the increasing scale of games development [1]. With the scale and scope of games development increasing, planning systems that are able to dynamically react at run-time are becoming sought after. Planning systems are excellent in linear games, such as F.E.A.R[2], which uses a GOAP Planning architecture to great effect.

A. Planners in Games

• STRIPS PLANNING

STRIPS Planning employs the use of states, goals and actions to discretise the world into a knowledge representation.

• GOAL-ORIENTED ACTION PLANNING

GOAP is a simplified STRIPS-like planning architecture that can be used within games. The GOAP architecture is modular, and allows for plans, actions and behaviours to be shared amongst agents [1].

B. STRIPS

The STRIPS planning system uses a number of components in order to build the knowledge representation of the game-state, in addition to determining the start and end goal-states. These components include;

• REPRESENTATION OF STATES

Planners are able to discretise the game world to build a knowledge representation of the world[3]. The planner takes information about the world and creates a series of conditions and facts, which are used to represent the current state [3].

• REPRESENTATION OF GOALS

The goal-state is the desired state that the planner wishes to end up in. The goal-state is defined by a number of facts, of which said facts are satisfied by actions taking place [3].

• REPRESENTATION OF ACTIONS

An action is defined as a set of preconditions that must be met in an action, in order for a resulting set of effects to be applied to the active planner state. An example action that can be used to initiate combat between agents can be:

\[
\text{Action}(a, tA),
\]

\[
\begin{align*}
\text{PRE} & : (\text{inRangeOf}(a, tA) \land \neg \text{notInCombat}(a) \land \text{isAlive}(tA)) \\
\text{EFF} & : \text{inCombat}(a, tA) \land \neg \text{notInCombat}(a))
\end{align*}
\]

Where a = instigator agent, tA = combat target

\[a \equiv \text{STanford Research Institute Problem Solver}\]
C. iThink

The iThink Library is a classical planning system written in C# specifically for the Unity game engine [4]. The iThink planner directly uses GameObjects in its literals to express facts about the world. The planner uses the same structure as the STRIPS planning system, which include initial states, goal conditions and action schemas using sets of positive literals/facts [4]. The benefit of using the iThink library is that it is purposely designed for the Unity3D[5] engine, in addition to being written in C#. Both of these factors are big positives, that allow for the quick implementation and deployment of the planning architecture.

D. Influence Maps

Influence maps are a technique that can be used to enhance decision making processes within games [6]. Influence maps hold a number of nodes, which correlate to a grid position in the world, or a set location.

Multiple influence maps can be used in a simulation, each with a different meaning. For example, there could be three influence maps being used with a planner; An obstacle map, a threat map and a resource map. Each of these three different maps will hold nodes that bear different meanings to their weighting. An obstacle map will tell the pathfinder that a node is blocked, a threat map will have a range of weightings which state which areas are dangerous, whilst the resource map will hold range of weightings which will influence the pathfinder to go to those locations. Multiple influence maps are able to be layered together, much like a lasagne, with each influence map being a layer, and the meat being the grid/planning model. Figure 1 shows the flow of transferring the influence maps to the planner.

In order to build the actions from the facts, iThink uses a schema system to define what actions are usable. An example schema that is used in this simulation is; GBActionMove-3-Tag::agent-Tag::moveLocation-Tag::moveLocation. The schema is made up of 3 sections. The first section details the name of the action to use, in this case it would be GBActionMove. The second component tells the iThink library how many parameters to expect. The third section lists the tags of the gameObjects that are to be used for each of the parameters. The iThink library utilises the tag system within Unity to differentiate GameObjects.

### TABLE I

<table>
<thead>
<tr>
<th>Fact Name</th>
<th>ParamTag</th>
<th>ParamTag2</th>
</tr>
</thead>
<tbody>
<tr>
<td>atLocation</td>
<td>agent</td>
<td>moveLocation</td>
</tr>
<tr>
<td>inCombat</td>
<td>agent</td>
<td>agent</td>
</tr>
<tr>
<td>notInCombat</td>
<td>agent</td>
<td></td>
</tr>
<tr>
<td>inRange</td>
<td>moveLocation</td>
<td>moveLocation</td>
</tr>
<tr>
<td>alive</td>
<td>agent</td>
<td></td>
</tr>
<tr>
<td>destroyed</td>
<td>agent</td>
<td>-N/A-</td>
</tr>
</tbody>
</table>

1The tag system is a way of grouping different objects under one tag, which can be used for sorting, collision or other means.

In the third section lists the tags of the gameObjects that are to be used for each of the parameters. The iThink library utilises the tag system within Unity to differentiate GameObjects.

### TABLE II

<table>
<thead>
<tr>
<th>Action</th>
<th>TagParams</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>moveLoc, moveLocation</td>
<td>atLoc(agent, moveFrom)</td>
<td>atLoc(agent, moveTo)</td>
</tr>
<tr>
<td>StartCombat</td>
<td>target</td>
<td>notInCombat(agent)</td>
<td>inCombat(agent, target)</td>
</tr>
<tr>
<td>StopCombat</td>
<td>target</td>
<td>inCombat(agent, target)</td>
<td>destroyed(target)</td>
</tr>
</tbody>
</table>

### III. IMPLEMENTATION

This section will be detailing how the iThink library was integrated into the test environment, furthermore this section will look into the models and methods required to get an implementation of iThink working and producing results.

A. Simulation Design

The simulation is designed as a small scale skirmish scenario. The core buildings are located in a static fixed location, with the player being able to define where the defences are. Once the user has defined the base, the planning system will devise a plan based on the only knowledge it has, which is the location of the core building.

The map itself is comprised of a number of movement locations. These are interconnected nodes that the planner is able to assign move actions to, with the lines stating which node can move to which, as seen in Figure 2.

![Connected Path Nodes](image)

Fig. 2. Connected Path Nodes

B. Planner Design

The planner has been Priorities that the planner will take to defeat the base. Any information on when it will route, and what facts could be added.

```markdown
<table>
<thead>
<tr>
<th>Action</th>
<th>TagParams</th>
<th>Preconditions</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>moveLoc, moveLocation</td>
<td>atLoc(agent, moveFrom)</td>
<td>atLoc(agent, moveTo)</td>
</tr>
<tr>
<td>StartCombat</td>
<td>target</td>
<td>notInCombat(agent)</td>
<td>inCombat(agent, target)</td>
</tr>
<tr>
<td>StopCombat</td>
<td>target</td>
<td>inCombat(agent, target)</td>
<td>destroyed(target)</td>
</tr>
</tbody>
</table>
```
C. Teaching the Planner

In order for the planner to be useful in an RTS environment, it needs to be constantly learning about the environment. A number of fact changes could occur, such as the destruction of a building, discovery of new buildings, hotspots of enemy locations and so forth.

The planner learns by linking into the influence map system, as previously detailed in Section II-D. When agents discover new information about the world, they feed it into a central influence map held by the AICommander. When a new plan is made, the information from the influence map is used to inject new facts into the planner, so that it becomes more knowledgable of the map.

D. Centralised vs Decentralised planning

The initial aim for this simulation was to have one AI Planner, which would ultimately figure out the plan based on all the knowledge available. The planner would then take the plan and split it up for each agent, so each agent in the simulation had their own itinerary.

One of the problems that was found with this approach was how unstable and poor at planning it was when taking multiple agents into account. When determining a plan for one agent to take down three target, it found a plan that worked and stuck to it. The problems started when introducing more agents into it’s fact list. Just the addition of one extra agent to the fact list was enough to throw a spanner in the works, which resulted in the planner being unable to find a plan.

To resolve this, the planner system was decentralized, with each agent in the AI team being given their own planner. By decentralizing, this meant that each agent was able to produce a plan for themselves, by only taking themselves into account. This removed the problem of additional agents in the planner breaking the planning phase. In addition, the decentralisation had a huge impact on the reduction of total actions. With 8 agents in the simulation, the original total number of available actions was 30,191, but after the decentralisation, it was reduced to roughly 1-2000 actions per agent. This dramatically improved load times, but it was still too slow when all agents were generating plans at the same time.

Further optimisation was made by creating a common pool of facts about the environment, which meant that when each agent started to make a plan, they pull a common factlist, which prevents having to search the world for facts each time.

IV. TESTING

This section looks into the process of testing the planner and putting it through it’s paces. For the testing, three initial targets have been setup that the planner knows about. There are then an additional 4 structures that have been added.

The focus of this paper is how the planner operates and plans accordingly based on it’s available, and new information, that it is aware of. As such, combat has been disabled, but the tanks are able to move about the environment as per the planner so that information may be learnt.

For testing, we will be using 8 agents in the simulation in a de-centralized planning approach. This will allow us to view the average time for an agent to generate plans, in addition to the maximum and minimum. We are also able to view how the time increase in relation to the number of actions available corresponding to the number of facts in the statespace. Five different sets of facts will be used, so we are able to gain an accurate view of the differences.

A. Generating Plans

The generation of plans works by giving the planner an initial state, which is the current state of the world. Then by giving the planner an end-goal state, the planner is able to determine the best route to solve the plan. The initial state will include all current locations of agent, path node connections, and locations of all enemy structures.

The benefit fo allowing each agent to individually generate plans means there is a boost in performance, and less of an impact when plans are generated.

B. Cost of Actions

<table>
<thead>
<tr>
<th>Total Facts (Actions)</th>
<th>Average Time</th>
<th>Max Time</th>
<th>Min Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>73</td>
<td>2656</td>
<td>24.5ms</td>
<td>41ms</td>
</tr>
<tr>
<td>79</td>
<td>2952</td>
<td>26.2ms</td>
<td>43ms</td>
</tr>
<tr>
<td>103</td>
<td>3592</td>
<td>40.7ms</td>
<td>84ms</td>
</tr>
<tr>
<td>133</td>
<td>4972</td>
<td>45ms</td>
<td>84ms</td>
</tr>
<tr>
<td>153</td>
<td>5064</td>
<td>48.3ms</td>
<td>88ms</td>
</tr>
</tbody>
</table>

One of the problems that was encountered with the iThink library was it’s requirement to completely rebuild the action set, even if as little as one fact had changed. The time it takes for the action set to build fluctuates based on the number of facts in the world. The majority of actions are move actions. On average, it takes 38ms to generate a set of actions, using 103 facts.
The time it takes to generate new actions is lower than expected, with 5,064 actions taking up to 88ms to compile, as seen in Table III. Although this is still a high number, initial expectations were to be much higher of this figure. The biggest source of problem is the time it takes to generate a plan. To generate a plan to destroy 3 targets, it can take at least 1500ms for a 10-step plan. This time is per-agent, and with 8-agents in the simulation, means the total plan generation time is roughly 12,000ms. From looking at Figure 4, we are also able to see the increasing time taken to generate plans on a per-agent basis when more facts and actions are introduced. This shows how un-scalable the system is, considering that given it’s current state, by the time the agent has generated plans and actions, other players may gain an advantage, to which the planner will constantly need to re-evaluate and re-plan.

V. Problems with Implementation

When implementing iThink into the simulation, there were some problems that surfaced throughout all stages, from installation to testing.

A. Reverse Engineering

One main issue that we came across while implementing the iThink library into the simulation was the severe lack, or existence, of legible documentation. The only documentation that exists is a BSc Thesis written by [4], which unfortunately is written in Greek. Due to being unable to translate the document, we had to explore and pick apart the iThink architecture. Unfortunately this took time to understand the system.

Furthermore, the latest version of the iThink library is unstable, with references to classes and unfinished search algorithms (including a new A-Star algorithm). The current version of iThink had to be salvaged from an example implementation of BlocksWorld, one of the sample projects listed on the website[4].

B. Performance

One of the main issues with using the iThink library is the performance cost it incurs. Even with a small game area with not many agents, or facts, about the environment, it can still take a noticeable amount of time to produce action lists and plans. Another problem occurs in regards to having the iThink planner learning about the gamestate. When new facts are learnt about the environment, ultimately new actions will need to be generated. Instead of simply generating a few new actions, iThink will re-generate the entire list of actions. With this happening each time something new is learnt, the expense is too much.

C. Scalability

Planning architectures in RTS games need to be scalable, due to a number of factors that can affect it, such as:

- **Number of Players**
  The more players that are taking part in a game, whether they are controller by a human or another AI, needs to be taken into account. Each player will add a new set of facts into the planning system, so planners need to handle it.

- **Size of Map**
  The size of the map, and the number of potential movement locations, is the biggest factor to take into account for planning systems. For each movement node, there are \((\text{connectedNodes} \times 2)\) possible actions, per agent, that needs to be taken into account. With a large number of movement nodes alone, the time it will take to generate a plan using iThink will be huge. Considering in Table...

D. Memory Usage

With a lot of actions & facts in the state, comes a lot of memory. Typically planners use a domain file which lists all of the preconditions, effects, parameters etc. On the other hand, iThink uses a class structure for each fact, action and so forth, so when a lot of facts and actions are introduced, the memory usage increases. With the decentralisation, this memory usage is set to increase further, due to each agent holding their own set of actions.

VI. Conclusion

This paper looked to achieve the following objectives.

- Implementing a STRIPS Planning architecture into an RTS environment
- Determine the feasibility of using planners in RTS games
- Determine if STRIPS planners can learn, and the impact of learning

In regards to implementing a STRIPS planning system into an RTS game, it is definitely achievable. With rich RTS environments though, there is a large amount of information that is needed to ensure that the planner is dynamic enough. Though the problems with implementation lead onto the second objective of feasibility of using a planner in an RTS game.

We recommend that if you are looking to implement a STRIPS planning system into an RTS game, that an alternative to the iThink library is sought out. The problem with finding other libraries is typically compatibility issues, with the majority of planners, such as JavaFF [7] being written in Java. This means that middle ware needs to be implemented in
order for the planner to communicate with Unity, or whichever desired engine is being used.

The impact that learning has on the planner is evident in figure 4, which shows that the amount of time taken to generate plans is exponential to the number of actions that have been generated per agent. As such, we can see that the planner is able to learn, but the impact of learning has a detrimental effect on the performance as a whole.

To conclude, we feel that although we were able to implement a STRIPS planning system into an RTS game using the iThink library, the amount of resources it requires, the lack of scalability and the amount of time taken to generate plans all point to the conclusion that it isn’t feasible to implement the iThink library into an RTS game, but however should seek an alternative.

VII. FUTURE WORK

There is a set of work that we would like to look into in the future. These include:

- **Improving Scalability of iThink**
  iThink currently lacks the efficiency to become scalable, and this is an area which could see a lot of work done. Optimizing the plan and action generation will be a big step towards making iThink scalable for RTS games.

- **Implementing "Hub-Planner" Systems**
  One feature that would be good to implement would be a hub-planner. This is essentially a mini commander script, that stores it’s own set of local facts. Each hub-planner is then connected to the core fact list, and is able to distribute information to a select number of specialist units. By implementing a system like this, we could effectively see ‘wing commanders’, or ‘captain’ units working with the general to coordinate and plan.

REFERENCES


