Game AI Multiagent Planning System

J. Angell
School of Computing
Teesside University
M2012929@tees.ac.uk

M. Cataldo
School of Computing
Teesside University
G7130816@tees.ac.uk

M. Chamberlain
School of Computing
Teesside University
M2014976@tees.ac.uk

Keywords: Game AI Multiagent Planning System, COM3036-N-BJ1-2015

Abstract

This paper discusses and evaluates the use of a planning system for a multi-agent, collaborative task, based on a simplified real time strategy (RTS) game AI. The system uses STRIPS style operators with a means-end analysis planner. There are a number of distinct agents that have specialised roles, namely harvester, collector and engineer. The effectiveness of the approach taken is evaluated and the performance of the solution is examined using a range of timed experiments. While the approach taken as part of this work has been shown to be effective, a number of optimisations and possible areas of development are suggested for future work.

1 Background

STRIPS (Stanford Research Institute Problem Solver) is an automated planner developed by (Fikes and Nilsson 1971). The planner uses means-ends analysis to search for operators that can be used to manipulate a world state to reach a state whereby a goal can be satisfied.

The current study makes use of operators that are similar to the ones found in the STRIPS planner. The operator style used in the current study is slightly more defined than the traditional STRIPS operator. In addition to the usual preconditions and postconditions, additional structures are present such as when, post, and achieves. This additional information is made use of within the planning engine and allows for more informed planning decisions, thus increasing the performance of the planner.

The definition of a world state is a requirement for planning mechanisms. The world state is fed into a chainer which will try to achieve a goal state by inferring which operators need to be applied. As the goal state is explicitly defined, it is sensible to use backwards chaining to create a plan. Forward chaining is better suited to situations where a large amount of data is available upfront or the goal is not yet known, new states or information are then inferred from this initial data. Although, a limitation of this approach is that it is difficult to ascertain whether inferred states progress towards the goal state. With backwards chaining, each operator placed on the stack will be necessary to achieve the goal state, assuming operators are protected properly. Thus making backwards chaining more appropriate for a goal-driven approach (Sharma, Tiwari and Keklar 2012).

2 Introduction

The scenario of the system is based on a simplified real time strategy (RTS) game AI; the likes of which can be found in games such as (Age of Empires 2016). This was selected because it is a well-known and successful example of a multiagent environment. For this scenario, three types of agent are used, each agent having a specialised role. The resource harvester's role is to locate and prepare resources. The resource is then collected by a resource collector whose role is to transport prepared resources to a base. The engineer monitors the base for available resources which are used to fulfil a given task such as building a bridge.

The SENTINEL system, as described by (Hendler 1992) uses truly distributed planning agents. Agents, or as they are referred to by Hendler, nodes, may have dependencies on other nodes in the sense that they may require assistance to complete a certain task. Tasks that require agent collaboration in this way lend themselves well to a multi-agent planning strategy, so it is important to consider this when building a planner for a collaborative task. As the problem domain of the current project requires multiple agents a backwards chaining planning mechanism is used. The system is multi-agent, but not distributed in the sense that each agent creates its own plans independently. This results in a system where agents do not run concurrently, each agent performs a singular operation, or operations, and all other agents are assumed to be idle.

A form of implicit agent communication is handled by a mechanism where after an agent performs an operation a specific tuple will be posted to the world state which may then trigger another agent into action. This means that the agents effectively communicate via the world state. A more explicit form of communication may be necessary in a concurrent multi-agent system where agents do not have a complete knowledge of the world state.

The planning mechanism will be visualised using NetLogo. The plan will be executed using Clojure; every time an operator is applied a command in the form of a textual representation of the operation is generated. NetLogo interprets the command list sent from Clojure and renders a visual representation of the task being performed.

3 Model

The model contains several types of tuple each describing the state of the world. We can categorise them into one of the three following areas, declarative, locative and state descriptive tuples. Two main types of declarative tuples are used (Figure 1). Firstly, an `isa` tuple is used to assign an id to an entity of a certain type. This structure allows for operator constraining via entity type. Secondly, a handles tuple is used to determine the resource states that an actor can interact with, effectively assigning their role as either a harvester, collector or engineer.

```
(isa h1 actor) (isa c1 actor) (isa e1 actor)
(isa t10 tile) (isa b1 base) (isa r1 resource)
(isa bh1 bhs) (isa l10 lake)

(handles h1 unprepared)
(handles c1 prepared)
(handles e1 stored)
```

Figure 1: Declarative tuples

The locative tuples shown in Figure 2 are used to describe where an entity is relative to another. An entity's locative tuple determines how it can interact within the world state depending on its current position.

```
(at h1 t10) (at r1 t11) (at l10 t12)
(on bridge t20) (at b1 t10) (holds h1 r1)
```

Figure 2: Locating tuples

Finally, the state descriptive tuples shown in Figure 3 are used to describe the state of an entity, this is used for resources which have three possible states, namely prepared, unprepared and stored. These tuples are used to control which operators can be applied to an entity, in this case a resource, depending on its descriptive state.

```
(unprepared r0) (prepared r1) (stored r2)
(goal t175)
```

Figure 3: Object state tuples

The modelled world is comprised of a collection of distinct locations, each with an id. These locations represent world locations called tiles. All other entities are placed in the world state relative to a tile, this provides a basis for locating and planning movement around the world space.

The model's operators adopt a hierarchical model for each agent. Every agent makes use of a set of generic operations such as move, pickup and drop. Additionally an agent has specialised operators based on their role within the scenario as outlined in Figure 4.

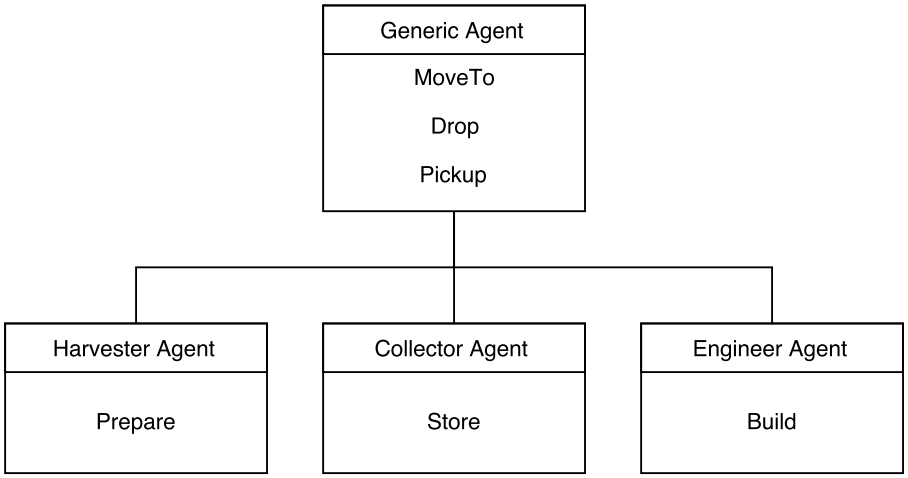


Figure 4: Agent operator hierarchy

The pickup operator shown in Figure 5 is an example the operator structure used as part of the planning system:

`:achieves` expresses the state that will be achieved after it is applied. The planner takes a goal and selects all operators that have a matching achieves statement.

`:when` has multiple uses, it can be used to obtain some matches that will be used later on within the operator, but it is mainly used to constrain an operator before it is added to the planners goal stack.

`:post` is used by the planner to add sub-goals which must be achieved before the operator can be applied.

`:cmd` produces a structured textual representation identifying the operator that was applied and some information on the matches that it used.

`:txt` generates a sentence explaining the outcome of the operator after it has been applied.

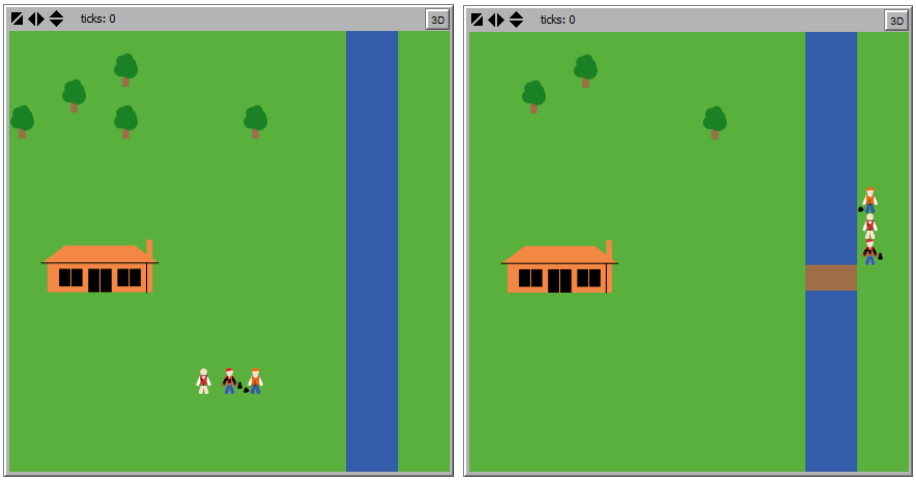
```
{:name pickup
 :achieves (holds ?actor ?r)
 :when ((at ?r ?t) (handles ?actor ?rs) (:not(unprepared ?r)))
 :post ((?rs ?r) (at ?actor ?t))
 :pre ((handles ?actor ?rs))
 :del ((at ?r ?t) (holds ?actor :nil))
 :add ((holds ?actor ?r))
 :cmd ((pickup ?r ?actor ?t))
 :txt (?actor picks up ?r from ?t)
}
```

Figure 5: Pickup operator

The world state and operators are wholly managed within Clojure leading to loose coupling with the model's presentation. The means-ends analysis planner found on (Lynch 2016), is utilised in conjunction with the initial world state and operator set to satisfy a list of

defined goals. On completion, the plan is translated into an abstract string format to be easily interpreted by the listening client, this is achieved using a matching mechanism.

This system uses NetLogo to provide a visual representation of a plan. To initialise the world state in NetLogo a series of commands are processed achieving an identical world state in both NetLogo and Clojure (Figure 6). The commands are derived from the initial world state in the form of tuples using the matcher.



(a) (b)

Figure 6: NetLogo model showing initial state (6a) and finished state (6b)

When a move command is received in NetLogo an A* search algorithm by (Singh 2016) is utilised to calculate a path to a desired location. The path finding algorithms adopts a mechanism of treating certain colour tiles as impassable, this is used for the blue tiles representing the river.

4 Testing

Testing has been carried out by changing various model properties and examining how execution time is affected.

The two most important factors of the planning system come down to execution time and appropriateness of plans. The quality of plans given by the system are quantified by the number of operators applied within the plan, assuming each operator is as easily applied as all others. As the problem domain is not of a scale that would have a significant impact on the quality of plans given by the system, testing is focused on execution time. Therefore, to determine the performance of the model in reference to the execution time, goal and world-size based testing strategies have been used. A similar approach is adopted by (Crosby and Petrick 2014) for a robot warehouse problem domain.

All experiments ran on a single machine with a 3.5GHz processor and 32GB of internal memory. Each test is repeated ten times and an average is taken to account for potential fluctuations in readings due to inconsistent load on the machine during the testing period.

4.1 Goal Based Testing Strategy

Figure 7: A chart depicting operator execution time

A goal based testing strategy is used to evaluate how the model performs for a number of problem scenarios. These ranged from small problems i.e. an agent moving to a desired location, to larger problem scenarios that involved full agent participation. The goal tuples used to represent each problem scenario have been selected according to the number of operators that would be required to achieve each goal. For example, a goal specification of `(at h1 t44)` would involve only a single operator, whereas a goal specification of `(on bridge t140)` would require the use of numerous operators. Therefore, altering the goal specification effectively dictates the size of the plan for each run of the experiment.

Figure 7 shows the average execution times associated with running each experiment for each of the problem scenarios. The rate of time increase is shown to be linear, however, varying gradients can be seen within the graph which suggest that some operators are more computationally intensive than others. Some operators attempt to match against the entire tile space so these are expected to take longer to complete.

4.2 World-size Based Testing Strategy

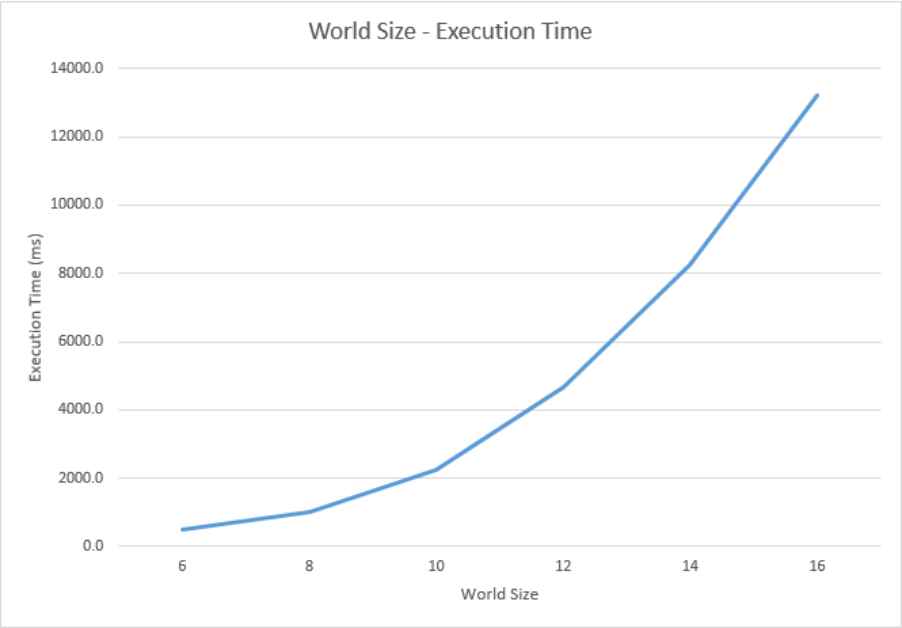


Figure 8: A chart depicting world-size execution time

Varying world sizes have been tested against time in order to ascertain whether introducing a larger environment would decrease performance. The findings show that increasing the word size causes an exponential increase in execution time. This is due to the fact that increasing the grid size causes an exponential increase in the number of tiles in the environment. So it follows that the rate of increase of both tiles and execution time is linear.

5 Evaluation

The main focus of this project was to develop a set of operators that would govern the behaviour of three distinct agents. These operators were then used in conjunction with a planning mechanism that comes together to form a planning system for a game AI. All proposed functionality was implemented successfully. Through introducing guard constraints, fewer specialised operators were needed to support the required agent behaviour.

The main limitation of the current model is its execution time, each tile has been explicitly defined so there is a large search space any time there is a match statement on a tile id. This is purely an issue with the world state tuple definitions. A more suitable approach would be to define locations only and let the visualisation client decide on where and how they appear within an environment. This approach does have some downsides in comparison to the current technique, one being the fact that some level of control over the world state is given up to the visualisation client.

Alternate approaches were considered for tuple design. A structure was considered where every tile and its neighbours would be modelled using a tuple design similar to `(connects t1 t2)`. The main motive for this approach was due to a different paradigm when constructing the move operator. It was thought that the move operator would enable a movement to an adjacent tile once per operator application, as opposed to delegating

the concern to NetLogo to simulate such a movement. This approach was dismissed as part of a greater decision to move location awareness away from the Clojure side as the consequent world state would have a significant impact on performance.

A prominent issue was encountered during development which involved the post-conditions of the `move-to-goal` operator. For this operator to be applied, some conditions would first have to be fulfilled by other operators, these conditions need to dictate that there is a clear path to the goal location. In cases where a bridge is required a variable number of post conditions are also required. Many approaches were considered to overcome this issue.

One solution is that an engineer is able to build the full bridge with one operator application, but this is not ideal as the requirements of the project specified that sometimes multiple resources should be utilised in order to progress towards building a bridge one step at a time. This is achieved by an engineer replacing a river tile within the environment to a bridge tile.

A recursive solution was considered whereby the `(on bridge)` tuple contained a reference to multiple tile identities and then passed as a goal state. The build operator would then be able to recursively deconstruct the tuple using the match expression `(on bridge ?t ??rest)` to match against the world state tuple `(on bridge t1 t2 t3)`. Running the planner using this match expression for the build operator proved to be effective, however, this approach lead to further complications whereby additional guard conditions were needed to prevent invalid tuples from being added to the world state.

Simply posting a fixed number of conditions that specify that a bridge is located between the agent and the goal location within the `move-to-goal` operator would solve the problem if the length of bridge was constant. However, to support the possibility of a varying length river, some preprocessing on the world state is carried out to ascertain how many bridge tiles will be required. The result of this preprocessing is used to post the correct number of conditions to ensure that a bridge will be constructed that spans the entire length of the river.

During the project further issues were encountered in regards to the tools used. The interoperation between Clojure and NetLogo gave rise to several problems. An issue related to thread blocking within NetLogo was encountered; as commands were being received the graphical updates were not being rendered. This may be an issue with the tool itself or more likely it is a consequence of using the technology in a way that it was not originally designed for. A better approach to achieve interaction may be to directly invoke NetLogo from within Clojure.

6 Future Work

The task discussed in this paper would undoubtedly benefit from a concurrent solution. Planning concurrently would allow agents to perform their individual tasks up to the point that some sort of collaboration is required. It is expected that concurrent application of operators would reduce the amount of time taken for the agents to construct a bridge in the scenario described in this paper.

Introducing a BDI model into the system could introduce conflict of interest between agents. If a scenario would arise where an agent could complete its task at the expense of another agent, then a BDI implementation may not yield a performance increase. This problem is not as prominent in non-concurrent systems as a singular goal is given to the planner and, assuming means-ends analysis is utilised, only operators that would progress the current state towards the goal state would be applied. With BDI, it is possible that

an agent has an intention to complete its small routine until it is no longer possible. This may introduce performance issues or even make some collaborative tasks impossible.

7 Conclusion

Overall the functional requirements of the project were met and a fully functional planning system was developed. Appropriate plans are given in a reasonable amount of time, however, the performance of the planner is hindered by the tuple structure that explicitly defines every tile. If this structure was redesigned then it is expected that the planner would perform in a significantly more timely manner.

References

Age of Empires: 2016, Age of empires: Play the greatest stories in history. Available at:. `http://www.ageofempires.com/`

Crosby, M. and Petrick, R.: 2014, Temporal multiagent planning with concurrent action constraints, *Proceedings of the ICAPS 2014 Workshop on Distributed and Multi-Agent Planning*

Fikes, R. and Nilsson, N.: 1971, Strips: a new approach to the application of theorem proving to problem solving, *International Joint Conference on Artificial Intelligence*

Hendler, J.: 1992, Artificial intelligence planning systems, *International Journal of Emerging Technology and Advanced Engineering*

Lynch, S.: 2016, Strips planner. Available at:. `http://s573859921.websitehome.co.uk/pub/clj/tools/planner.htm`

Sharma, T., Tiwari, N. and Keklar, D.: 2012, Study of difference between forward and backward reasoning, *International Journal of Emerging Technology and Advanced Engineering*

Singh, M.: 2016, Netlogo user community models: Astardemo1, *Proceedings of the ICAPS 2014 Workshop on Distributed and Multi-Agent Planning*