Anytime Planning For Agent Behaviour

Nick Hawes
School of Computer Science, The University of Birmingham

Abstract

For an agent to act successfully in a complex and dynamic environment (such as a computer game) it must have a method of generating future behaviour that meets the demands of its environment. One such method is anytime planning. This paper discusses the problems and benefits associated with making a planning system work under the anytime paradigm, and introduces Anytime-UMCP (A-UMCP), an anytime version of the UMCP hierarchical task network (HTN) planner [Erol, 1995]. It also covers the necessary abilities an agent must have in order to execute plans produced by an anytime hierarchical task network planner.

1 Introduction

The overall aim of the work discussed in this paper is to produce a method for generating behaviour for an agent in a computer game, or other forms of interactive entertainment. There are two challenges inherent in designing any AI system for such an application. The first is the behaviour of the environment the agent will typically be placed in. The majority of computer game worlds are highly dynamic, complex and unpredictable. The second is the limited amount of processing time available for AI processes (usually around 10% of total CPU time). This is because the vast majority of processing time in games is dedicated to other facets of the program such as graphics and monitoring user input.

When generating plans for agents to follow in an environment such as a computer game, we are not interested in producing plans that are optimal. Ideally a plan will allow an agent to act out the correct behaviour for a situation, whilst appearing believable as a character. This may sometimes include displaying weaknesses (after all, how many humans can produce a perfect plan when under pressure?). For this reason the work discussed in this paper will not be primarily concerned with optimality or completeness, but with just generating behaviour.

In [Hawes, 2000] I discussed the use of a planner based on the anytime paradigm [Dean and Boddy, 1988]. In brief, an anytime algorithm is an algorithm that can be interrupted at any time, and the quality of the result that will be returned after an interruption increases with processing time. The primary advantage of an anytime planner is that the agent using it is afforded complete control over the planning process. For example, if the agent came under threat whilst planning, it could interrupt the planning process and store the plan to execute, or to continue elaborating later. Using an anytime planner also allows the agent to reason about, and execute trade-offs between the time taken to create a plan and the expected gains of executing the plan.

2 An Anytime Planner

An anytime planner is a planner that is constrained to behave in a certain manner. The constraints are laid out in [Zilberstein, 1996]. From these constraints, two main problems can be identified; interruptibility and quality.

The interruptibility problem is defined as "The algorithm can be stopped at any time and provide some answer" [Zilberstein, 1996]. The ability to interrupt an anytime algorithm is the primary appeal of using such a technique. Being able to stop a planner allows the agent controlling it to optimise its use of (limited) processing time by preventing the unnecessary elaboration of a plan (if it is already of a high enough standard), or by stopping a process that is planning for a goal that has been made irrelevant or serendipitously achieved in the environment. Being able to retrieve a plan from a planning process at an arbitrary point in time allows an agent to keep up with dynamic and unpredictable worlds (such as those found in computer games).

The quality problem is really two related problems; how can we measure the quality of any plan (at any stage of completion) that the agent can produce, and how can we ensure that the planning process is monotonic (the quality of solutions does not decrease with time)? We require the planning process to be monotonic with respect to time in order to allow the agent to safely perform a trade-off of processing time against plan quality. Measuring the quality of plans is necessary both to allow us to judge whether the planner is monotonic or not, and to allow the agent to monitor the planning process and estimate the utility of a particular plan.

The following sections will introduce the planner being used as a basis for the anytime planner, and then discuss how the aforementioned problems have been approached.

3 The A-UMCP Planner

The Universal Method Composition Planner (UMCP) [Erol, 1995] is used as the basis of the anytime planner (Anytime-UMCP) discussed in the remainder of this paper. The reason a hierarchical task network planner was chosen is discussed in the following section. The reasons why the UMCP planner was specifically chosen are that it has been well documented allowing for easy understanding and modification, and that source code for a version of the planner is freely available.

4 The Interruptibility Problem

The requirement of being able to to interrupt the planner is not a difficult problem, as this can be done fairly simply (since most planners work in discrete cycles). It is the answer returned after the interruption which causes the problem. For different planning approaches, interruptions will result in different outcomes. For instance, interrupting a forward chaining total order planner will result in a plan fragment that would take the agent from its start state (the initial condition), to some other point in the environment/search space. There may not be any guarantee that this point is any closer to the desired goal state than the start point was. If a total order regression planner was interrupted, the resulting plan would be a plan from an arbitrary point in the search space, to the goal state, and there would be no guarantee that the agent could get to that point. The outcome would be different again for a partial order planner. Because these results would be almost useless when generating behaviour for an agent, an HTN planner has been used as the basis for an anytime planner.

An HTN planner works in cycles reducing an abstract method until it is expressed completely in primitive (executable by the agent) operators. After every reduction, the resulting task network is an entire plan at varying levels of abstraction. At any point during the planning process, the current (partially complete) plan being operated on by the planner can be considered to be a solution to the planning goal *at the current level(s) of abstraction*. This means that although further refinements of the partial solution may reveal inconsistencies that prevent the plan achieving the goal, at this level of abstraction the plan is capable of achieving the goal. From this we can infer that if an agent can execute the partial solution at this level of abstraction, then it can achieve the goal.

4.1 Executing Abstract Plans

This then is the real research problem: how can an agent execute a plan that contains abstract methods? The ability to do this would allow an agent to use the results from an interrupted HTN planner. Currently there is no elegant answer to this problem. All agents that aim to act intelligently in a dynamic environment must have a library of reactive actions and basic reactive plans to enable them to deal with situations requiring reactive control. If this library was extended to equip the agent with reactive implementations of the abstract methods in its world, it could then execute plans containing such actions. There are a number of problems with this approach. Firstly, if the agent had a large number of possible abstract methods it could lead to a storage and indexing problem. This could be avoided by taking advantage of the fact that there will only be a limited number of primitive actions that the agent can actually perform in the environment, and all of the abstract methods must be constructed from these. A network could be built using the refinements contained in the HTN reduction information, then an abstract method could be translated into executable primitives using this network.

A bigger problem with this method of interpreting abstract methods is that abstract methods in HTN planners typically have more than one possible refinement. One common alternative refinement in the UMCP planner is the dummy or 'do nothing' action, that allows an abstract method to be reduced to nothing if it is already true. A possible approach to dealing with this would be to interpret the abstract method in a teleological manner [Nilsson, 1994], so that if a goal is satisfied it is passed over by the action executor. Unfortunately this approach will not be applicable if two distinct ways of refining an abstract method are possible. In this case the most widely applicable action could be used, or some simple checks could be added as a heuristic to encourage the use of the correct action. It is important to minimise the processing required to interpret abstract methods because when an agent is doing this it should appear as if it were just executing primitives. This is desirable because when planning is interrupted, the agent will be expecting a plan to execute immediately, and not one which requires additional processing.

5 The Quality Problem

5.1 A Quality Measure for A-UMCP

Before looking at how we can measure the quality of any partial plan produced by UMCP, we have to make explicit what is meant by 'quality'. Plan quality can be measured by a variety of factors. Measurements can include number of steps, use of resources, the time necessary to execute the plan, or the number of constraints satisfied. In each of these cases the quality measure is chosen because of what we are trying to optimise in the final plan. For example, finding a plan that uses the smallest number of steps will require a quality measure that involves reasoning about the number of steps. In the case of an anytime planner the choice of quality measure is dictated by the fact that the planning

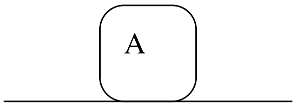

Figure 1: Cost of Clear(A) = 0

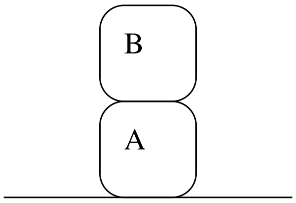

Figure 2: Cost of Clear(A) = 1

process can be interrupted, and the resulting plan will be interpreted by a process that could possibly add errors into the result. The method previously discussed for interpreting abstract methods has an inherent weakness that the more abstract the method being interpreted, the more likely it is that an error will be introduced during execution, or at least some inefficiency will occur (due perhaps to a failure to exploit possibly better orderings of actions). Because of this, the proposed plan quality measure is based on how abstract the plan is; a more abstract plan is costlier because it is less likely to be executed properly after an interruption occurs.

To implement this quality measure, we need a heuristic for judging the 'abstractness' of plan steps (in this case abstract HTN methods and action primitives). The most direct way to measure this is to count the number of refinement steps it will take to transform an abstract HTN method into a set of primitives, a calculation that could be performed before runtime (as the set of possible refinements is static) and then stored for lookup during the planning process. Unfortunately, although the allowable refinements of a method are static (i.e. this method will always become this or that series of operators), different initial situations can lead to wildly different cost measures for the same method. For example, in the Blocks World operators provided with the UMCP planner, the goal of clearing the top of one block (`A`) is expressed as `Clear(A)` (which represents both an abstract method to make `A` clear, and the atom to be eventually added to the world state). This method can be reduced into either `DO NOTHING` (in the case that `A` is already clear) or `Clear(?x) -> unstack(?x,A)` (in the case where some block `?x` is stacked on `A` and must be made clear before it can be unstacked from `A` onto the table). The recursive nature of the second possible reduction enables a variety of situations to be easily expressed in HTN notation, but prevents the possibility of a pre-calculated measure of 'abstractness'. This is because the same method can represent different amounts of action and consequently different amounts of abstraction.

An example of this can be seen in the following situations. In Figure 1, `A` is clear so `Clear(A)` requires no refinement (except replacement with a dummy action), and hence can be given a cost of 0 (i.e. not abstract). In Figure 2, block `A` is covered by one other block (`B`), `Clear(A)` requires one reduction into `Clear(B)` (which is already clear) `->` `unstack(B,A)`, and hence can be assigned a cost of 1. This is based on the number of reductions away from being all primitive, with `Clear(A)` having a cost of 0. In Figure 3, `A` is covered by `B` and `C`, and working from our previous results `Clear(A)` can be awarded a cost of 2. This demonstrates that it is not always possible to assign a fixed cost to abstract operators.

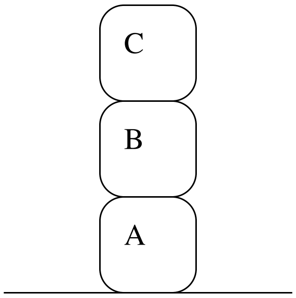

Figure 3: Cost of Clear(A) = 2

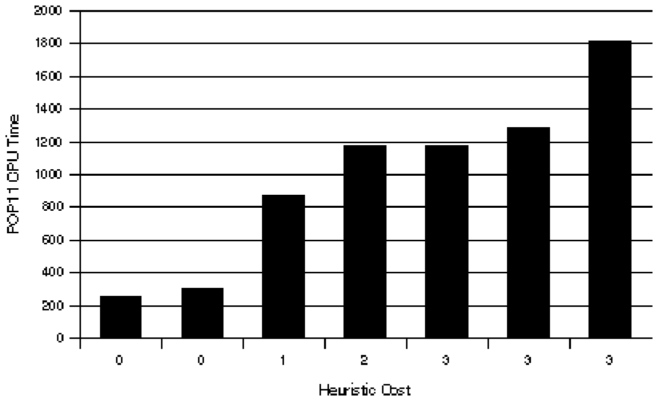

Figure 4: Comparison between heuristic cost and actual cost

5.2 Applying the HSP Heuristic to HTN Planning

As we have already seen, the UMCP notation allows abstract operators to be described in terms of the eventual atomic goal of their refinement (e.g. the actions required to get `A` onto `B` can be expressed as `On(A,B)`). This, taken with the operators that represent the primitive actions available in the planning domain (e.g. `unstack(?x,?y)`) and a representation of the current state, comprise a state-based planning problem. In this framework a heuristic for estimating the cost of achieving a goal has already been developed. The heuristic is based on work presented in [Bonet et al., 1997] and is used in the Heuristic Search Planner (HSP) of [Bonet and Geffner, 2001]. The cost of achieving a goal in a particular state is calculated by considering a relaxed planning problem in which the delete lists of operators are ignored. The heuristic works in cycles, adding the effects of all applicable operators to the current state. The number of cycles it takes before the desired atom appears in the state is the heuristic cost of achieving it, given the particular set of atoms used as the initial state.

Although the HSP heuristic does not deal directly with reductions of abstract methods, the number of operator applications required to achieve a goal is directly related to the cost of refining the associated abstract method. This is because the majority of processing time during refinements is taken up with propagating the constraints associated with each operator. The more operators that are needed to achieve a goal, the longer the time taken to process the constraints, the higher the cost. A comparison of heuristic cost estimates (using the aforementioned heuristic) and actual processing costs (measured in CPU time) confirms this. The results of the comparison for some sample Blocks World problems are shown in Figure 4. This shows that the heuristic is informative (i.e. it can correctly distinguish between tasks of different abstraction levels), but is not always totally consistent (tasks with the same heuristic cost can have varying actual cost).

Goal-based abstract methods (i.e. `On(A,B)`) are not the only type of action that can be present as a step in a UMCP plan. This means that in order to have a quality measure for entire plans, we need to develop cost measures for other types of step as well. Of the other types of step the simplest is the 'do nothing' step. Because this step involves doing nothing, it is always assigned a cost of zero. Action primitives are the next type of possible step we must assign a cost to. Because they are not abstract, logically we should assign a zero cost to them, but in domains in which it is necessary to minimise the number of plan steps they can be assigned a cost of 1 to penalise plans that use more primitive actions than other similar costing plans. If this is done, then all abstract method costs must be increased by 1 so as to distinguish between a primitive, and an abstract method that would originally have had a cost of 1. Based on this way of costing individual actions, the total cost of a plan is simply the sum of the cost of the individual steps.

5.3 Monotonicity In A-UMCP

Once we can measure the quality of individual plans, we can address the issue of making the quality of the result returned by the planning process monotonic with respect to time.

The A-UMCP planner produces plans using a search strategy that picks the partial solution with the lowest cost, then refines it by either reducing an abstract method or enforcing constraints associated with previously reduced methods. The planner stores all of the partial solutions in a list sorted by cost (the open list), and it is the head of this list which is picked for refinement. If an interruption occurs it is also this plan that should be returned as the solution, since its position at the head of the list means that it is the current 'best' plan. Because of this, if A-UMCP is to be monotonic it is the quality of the plan at the head of the open list that may never decrease as planning time increases.

Unfortunately, in HTN planning it is not possible to accurately determine whether a certain partial solution will lead to a dead-end or not, before it has been refined to the point at which the errors become apparent, The result of this is that it is impossible to produce a guaranteed monotonic anytime planner using just the simple search setup mentioned above. This is because the planner may inadvertently follow branches in the search space which lead to dead-ends, and it will then have to start searching again from a point of possibly higher cost/lower quality. To make the anytime planner appear monotonic to the agent controlling it, a partial plan with the highest quality yet encountered needs to be stored separately from the current plan. This stored plan can then be returned if an interruption occurs. Every cycle, the planner must compare the cost of the current head of the open list to that of the stored best plan. If the quality of the head of the list is higher than any other plan yet encountered, it is stored as the current best plan. Storing a previous result to make an anytime algorithm appear monotonic has been used previously in route planning [Zilberstein and Russell, 1993]. They stored a more (physically) abstract result from a previous route planning process to return if the subsequent, more detailed, route planning process was interrupted.

5.4 Using the Cost Measure as an HTN Critic

Critics are used in HTN planning to recognise and prune partial solutions that lead to dead-ends in the search space [Erol et al., 1995]. The UMCP planner uses powerful logic-based critics to determine whether a partial solution contains any inconsistencies that will prevent it from being refined successfully. These critics are domain independent and have no knowledge about the actual meaning of the plans. Because of this they fail to prune a commonly occurring reduction that introduces unsatisfiable plans into the search space. This refinement is the introduction of the dummy or 'do nothing' operator.

Any abstract method that can be true in the planning domain (e.g. `Clear(A)` or `On(A,B)`) must have a possible reduction to the 'do nothing' operator. Every time an abstract method is reduced, an extra branch is added to the search space representing this 'do nothing' situation. These regularly represent dead-ends because the necessary atom is not true in the current state. Standard UMCP has to refine each of these dead-end tasks further to determine whether the 'do nothing' action is allowable, whereas A-UMCP does not. During the reduction process, if an abstract method has a heuristic cost of 0, it means that it is already true in the environment (0 operator application cycles are required to introduce it into the environment). If it has a cost greater than 0, it is not already true. In the latter of these cases, the 'do nothing' reduction is pruned from the search space, resulting in the planner having fewer dead-ends to pursue.

5.5 Using the Cost Measure as a Heuristic

Because the UMCP planner has inbuilt critics which remove any partial plan that will fail to lead to a solution, any search heuristic developed to work with the planner need only guide the search through the space of possible solutions. The ideal behaviour for the planner is to produce, as quickly as possible, a plan which can be interpreted successfully by the agent (this relates to another of the desired properties of an anytime algorithm; diminishing returns [Zilberstein, 1996]). To do this, the heuristic guides the planner to refine the partial solution which appears to be the closest to being entirely primitive (i.e. the cheapest plan using the previous cost measure). This 'greedy' approach may not always lead directly to a primitive solution, but should quickly provide the agent with a plan that can be interpreted with some degree of confidence (i.e. one that is considerably less abstract than the initial condition).

The heuristic currently being used to guide A-UMCP is based on weighted A* [Pearl, 1984], and is expressed as

$$f(tn) = g(tn) + Wh(tn)$$

where $g(tn)$ is the accumulated cost of the task network (the plan), $h(tn)$ is the heuristic estimate of the cost of reducing the task network into a completely primitive state, and W is a weighting to make the search 'greedier' (currently the value 4 is being used).

5.6 Results

Currently only the A-UMCP planner has been finished. The agent that will use it is still being implemented. Figures 5 and 6[1] show how the cost of the stored 'best' solution decreases monotonically with time. The results also demonstrate how the planner makes large reductions in cost quite early on (usually within the initial third of the cycles) in the planning process, with the plateaus across the later cycles representing changes to the plan that don't involve reduction to less abstract levels. This, by definition, does not effect the measure of plan quality.

The behaviour of the planner is such that it tends to choose a branch of the search tree, and follow it until it either becomes pruned from the search space (i.e. it is not a feasible solution) or is found to lead to a solution. This lack of deviation is what provides the early gains in quality made by the planner. If this early search beam does not lead to a solution, it will at least leave a stored partial solution that is less abstract, and hence more easily executed than the rest of the available partial

[1] Domain adapted from the shopping domain presented in [Russell and Norvig, 1995] Chapter 8.

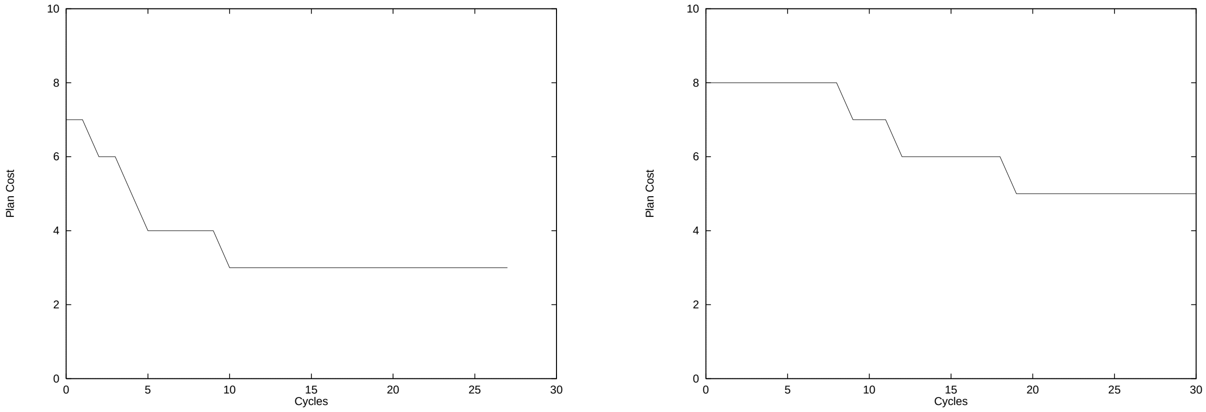

Figure 5: Blocks World example performance profiles for 3 and 4 block problems

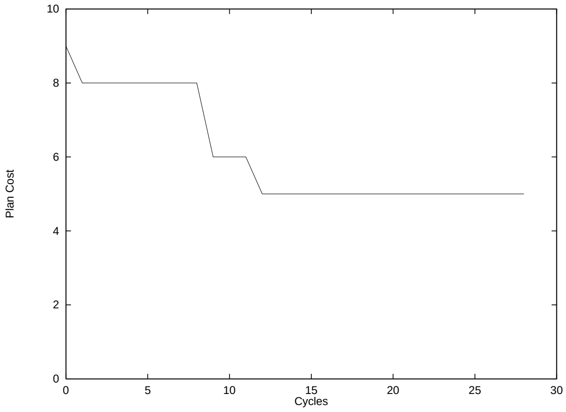

Figure 6: Shopping World example performance profile for 3 items and 3 locations

solutions. This will be the solution that is returned if an interruption occurs during the subsequent search within costlier areas of the search space.

This behaviour can be seen in Figure 7. The charts depict what percentage of nodes expanded to find the first solution were on the search branch leading to that solution. A value of 100% means that the search has not deviated from its initial choice of branch and has found a solution on that branch. A lower number means that the planner has deviated a lot more, partially expanding different branches of the search tree before finding a solution. At worst the results of using the heuristic are no better than not using it. At best it provides an improvement of around 60% over the non-heuristic version[2]. In terms of an agent using A-UMCP, a higher percentage means that the agent will have access to a less abstract (i.e. better) plan earlier in the planning process, because more time has been spent refining a single solution.

6 Conclusions and Future Work

The first part of this paper introduced and discussed some of the necessary requirements for constructing a planner based on an anytime algorithm. From the results it can be seen that these requirements have been met. Anytime-UMCP provides a monotonic increase in plan quality as

[2]Measurements for the non-heuristic version are included as a point of reference, but increasing upon these values should be a trivial task for any heuristically guided planner.

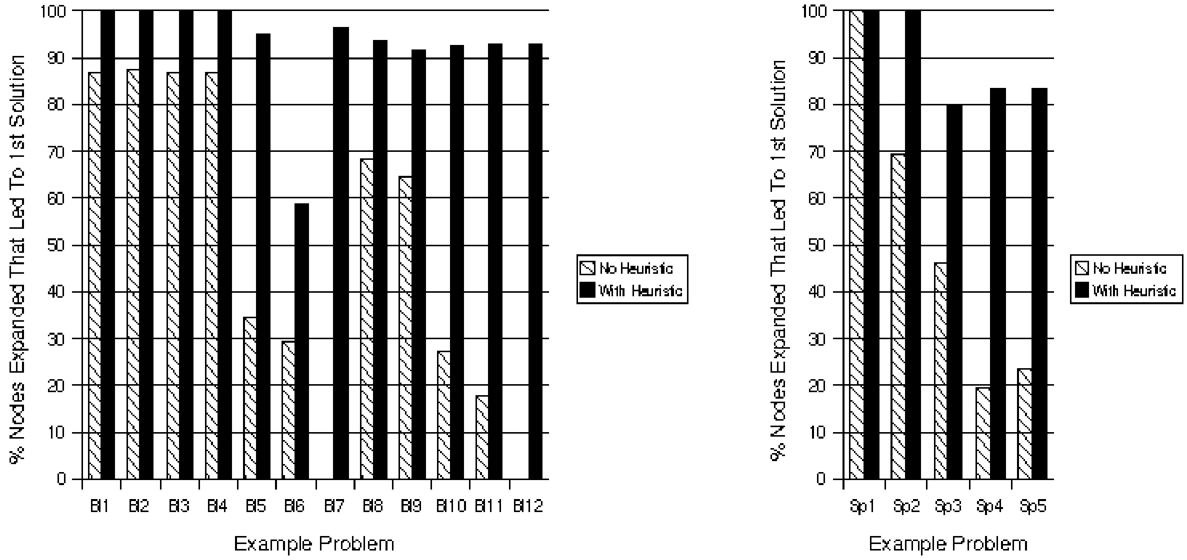

Figure 7: The effect of the heuristic in the Blocks (left) and Shopping (right) worlds

processing time increases. The quality measure presented in this paper involves heuristically estimating how abstract the plan is, and hence how successfully an agent (with the necessary abilities) could execute it. The quality measure used for partial solutions is really the most crucial aspect of implementing an anytime algorithm. Without it, the other algorithm requirements cannot be evaluated, and there would be no way to judge its performance. Because of this, quality measures for partial plans, particularly when the plans are to be executed by an agent, is an area that requires extensive future research.

In this research, it is the topic of how partial plans can be executed that has produced the least satisfactory answers. The issue of how an agent can execute a plan containing arbitrarily many abstract methods has been addressed, but no definite conclusions have been reached. Having a reactive action for each abstract method appears superfluous, so perhaps the information stored in the reduction schema for the HTN planner may be used to make this more efficient.

The real test of this research will come when an agent has been implemented that uses A-UMCP to generate the majority of its behaviour in a dynamic and complex world.

7 Acknowledgements

This research is supported by sponsorship from Sony Computer Entertainment Europe. This research has also been supported by Aaron Sloman in the form of advice, critical assessment and supervision.

References

[Bonet and Geffner, 2001] Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence: Special Issue on Heuristic Search*, 129:5–33.

[Bonet et al., 1997] Bonet, B., Loerincs, G., and Geffner, H. (1997). A robust and fast action selection mechanism for planning. In *Proceedings of The Fourteenth National Conference on Artificial Intelligence*, pages 714–719.

[Dean and Boddy, 1988] Dean, T. and Boddy, M. (1988). An analysis of time-dependant planning. In *Proceedings of The Seventh National Conference on Artificial Intelligence*, pages 49–54.

[Erol, 1995] Erol, K. (1995). *Hierarchical Task Network Planning: Formalization, Analysis, and Implementation*. PhD thesis, Department of Computer Science, The University of Maryland.

[Erol et al., 1995] Erol, K., Nau, D., Hendler, J., and Tsuneto, R. (1995). A critical look at critics in htn planning. In *14th International Joint Conference on Artificial Intelligence*, pages 1592–1598.

[Hawes, 2000] Hawes, N. (2000). Real-time goal-orientated behaviour for computer game agents. In *Game-On 2000, 1st International Conference on Intelligent Games and Simulation*, pages 71–75.

[Nilsson, 1994] Nilsson, N. J. (1994). Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158.

[Pearl, 1984] Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley.

[Russell and Norvig, 1995] Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach*. Prentice-Hall International, Inc.

[Zilberstein, 1996] Zilberstein, S. (1996). Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83.

[Zilberstein and Russell, 1993] Zilberstein, S. and Russell, S. J. (1993). Anytime sensing, planning and action: A practical method for robot control. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1402–1407.