Goal planning for automated agents

BY

Michael Dawe
B.S., Computer Science, Rensselaer Polytechnic Institute, 2002
B. S., Philosophy, Rensselaer Polytechnic Institute, 2002

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the graduate studies program
of DigiPen Institute of Technology
Redmond, Washington
United States of America

Fall, 2007

Oral defense signature page goes here

Thesis approval signature page goes here

Contents

1 Acknowledgments **8**

2 Problem statement **9**

3 Planning overview and early history **10**

4 Variations on planning algorithms **12**

4.1 Satisfiability and planning 12

4.1.1 Differences between deduction and satisfiability 12

4.1.2 Summary of results 13

4.2 Planning with graph analysis 14

4.2.1 Planning graph structure 14

4.2.2 Summary of results 16

4.2.3 Graphplan for multiple agents 16

4.3 Applicability to games 17

5 Parallelizing plans with partial-order planners **17**

5.1 When actions can be performed in parallel 17

5.2 Types of parallel plans 18

5.2.1 Independent actions 18

5.2.2 Independent actions relative to a goal 18

5.2.3 Independent subplans relative to a goal 18

5.2.4 Interacting actions 19

5.3 Partial-order planning with interacting actions 19

5.4 Applicability to games 20

6 Plan recognition **21**

6.1 Plan recognition overview 21

6.2 Planning and plan recognition 21

6.3 Applicability to games 22

7 Plan merging **22**

7.1 Positive goal interactions 22

7.2 Avoiding resource conflicts 25

7.2.1 Applicability to games 26

7.3 Operator merging 27

7.3.1 Applicability to games 27

8 Hierarchical Task Network planning **28**
8.1 Overview of HTN planning 28
8.2 Applicability to games 30

9 Applications of planning in games **30**

10 Proposal for research **32**

11 Research products **33**
11.1 Planning system 33
11.1.1 Search algorithm 33
11.1.2 Goal classes 34
11.1.3 Action classes 34
11.1.4 Plan class 35
11.2 World representation 35
11.3 Agent definition 37
11.4 The Planning Algorithm as used in an Agent 37

12 Improving the planner **38**
12.1 Scheduling 38
12.2 Goal merging 39
12.3 Plan merging 42
12.3.1 Implementing a plan merging algorithm 43
12.3.2 Beyond single-agent merges 44
12.3.3 Strategies for improving action searching 45

13 Conclusions and future work **46**

List of Figures

1 Operators and the generated Planning Graph for a particular planning problem. Taken from [3]. Dots in action layers represent no-op actions. 15
2 An example goal-plan tree. From [25]. 23
3 Symbols and terms used in Figure 2. From [25]. 23
4 Basic HTN planning algorithm. From [7]. 29
5 An agent's logic to determine the most important goal for planning purposes. 38
6 Algorithm to place actions into the correct order based on scheduling precedence relations. 40
7 Algorithm to merge two goals into a single goal. 41
8 A plan merging algorithm with a known action to merge on. 45

1 Acknowledgments

At DigiPen, Dr. Bikram Banerjee served as a frequent sounding board for ideas and consistently asked the right questions to make sure I understood every paper I read. Dr. Dmitri Volper also provided useful feedback and served as chairman of the thesis committee. Dr. Xin Li provided much encouragement as well. Dr. Michael Jahn and Dr. Rania Hussein were both kind enough to serve as committee members. Steve Rabin provided insightful comments and enough of a push to have me write up a similar idea for a book. Finally, the idea for this work was inspired by Jeff Orkin, who was always provided thoughtful, polite, and timely responses to all my questions.

Abstract

Goal planning is a well-researched method for automated agents to formulate a plan of action to solve a problem. This paper presents an overview of the development of planning in the general artificial intelligence setting and specifically its recent applications to reasoning for artificial agents in real-time games. Additionally, the author presents research on plan merging algorithms as applied to game applications.

2 Problem statement

Real-time simulations and games in particular have demanding technical requirements as well as high consumer expectation for delivering the best possible experience. Artificial intelligence is one area of game production that is particularly demanding in a wide variety of ways: programmers must have their agents interact with increasingly complex environments, exhibit myriad different behaviors, and interact intelligently with the player and other agents. As games and hardware grow more complex, so must the AI systems designed to control those agents. Yet complexity is increasingly a problem for both designers and engineers as more complex systems have more subtle interactions and the possibility for unintended behavior. Indeed, several major talks at the Game Developer's Conference in recent years have dealt with handling complexity in artificial intelligence systems for games [21], [12].

Traditionally, game systems have utilized finite state machines (FSM's) to control character behavior. In part, FSM's are widely used because they are widely used; the collective knowledge about FSM's is vast from both practical experience and academic research, and the concept behind them is easy to understand. However, state machines have some disadvantages as well. As finite state machines grow larger, they also become more difficult to understand, making accurate prediction of their behavior under a variety of circumstance nearly impossible. Larger FSM's take up more space in memory and are more challenging to design consistent behaviors for as well. Additionally, and especially as projects become larger, it is desirable to be able to design behaviors for characters and reuse those behaviors for other characters. Yet in the past, reusing FSM's for other characters has proved difficult or impossible [21].

Why is reusing a finite state machine so difficult? It should be simple enough to write a FSM for a particular behavior, such as "Attack," or "Chase," and reuse that particular FSM for any character that needs to perform those actions. The difficult part comes when designers wish those behaviors to be slightly different for different characters. The FSM's become much larger as the behaviors incorporate special logic, and

tweaking or changing one part of the FSM means extensive testing. Each tweak could have unintended effects on the behavior, or even other behaviors, often in unexpected ways. This risk is often unacceptable, particularly as the project cycle ends.

Jeff Orkin has suggested planning as a way of overcoming these obstacles in game design [21]. The motivation for examining real-time planning for artificial agents in games came about from a desire for characters to have atomic, reusable actions that can be easily shared between characters while providing more complex and realistic behaviors with reduced complexity.

3 Planning overview and early history

Planning in artificial intelligence was developed as an alternative way of looking at problem solving through a search of states. Traditional searching algorithms, such as breath-first search or depth-first search, are used to search through a set of states in search of a target state, optionally using a heuristic function to help guide the search. However, such searching algorithms can only blindly follow the heuristic while searching for the target goal state. Goal planning systems were developed to help agents reason about how they were to accomplish their goals.

Russell and Norvig [23] present an example concerning an agent with a shopping list: the agent is at home, and has a number of items it needs to acquire (say, a hammer and a quart of milk). Traditional searching algorithms could search through a significant number of states in their searches for the goal state (being at home with the hammer and milk). Such searches are mostly blind, considering equally all possible actions, including going to sleep, reading a book, or going to the hardware store, when in actuality only one of those actions holds any hope of being fruitful for the agent. Planning systems attempt to codify information about the actual actions an agent can take in order to help the agent think and act rationally about the problem given.

Planners usually represent goals (a target world state) and actions (ways for the agent to change the world state) using some derivative of first-order logic or another formal language. Indeed, the first major planning system, STRIPS, uses a form of predicate logic to represent the state of the world and actions an agent can perform [16]. STRIPS was originally written for a robot named Shakey, a robot at the Stanford Research Institute[1] in the early 1970's. Since then, STRIPS has become a standard for planning languages in numerous projects.

Most planners have some terminology in common. An planner is given a goal, an

[1] Now SRI International.

initial state, and a set of actions to use in accomplishing the goal. A *goal* is quite simply a target world state: the state of the problem as the agent wants it to be. In the example above, the goal was for the agent to be at home with a hammer and a quart of milk. The *initial state* is the starting point of the search, or the world state as it is currently. Using the above example again, the initial state might just be the agent at home. It may, of course, include other things, such as the fact that the agent has some bananas, although such things will be irrelevant to the planner. *Actions* or *operators* are the things an agent can do in order to change the state of the world. Some possible actions for our example problem would be buying an item or moving to a different place in the world. Many planners allow for actions to have variables, allowing the planner to have a single action such as *Buy(x)* for all possible values of *x*. An action can have *preconditions* and *effects*. For instance, a precondition of *Buy* might be having enough money to buy the item. Effects of actions explain to the planner how the action changes the world state; *Buy(x)* has an effect of the agent now possessing the item *x*. It would also specifically state other consequences of the action, such as the agent having less money than it did before the action was executed.

With these definitions, planners can then examine problems to come up with *plans*, a formalized solution to the problem. A plan consists of an ordering of actions the agent should take to achieve the goal state. Plans should also contain variable bindings for actions with variables. Plans can be *partially-ordered*, where some actions may be left unordered relative to each other, or *totally-ordered*, where all actions are given in a specific execution order. Both types of planners output a set of ordering constraints, indicating which actions must be performed before any other particular action. Partially-ordered planners are more widespread for a variety of reasons. First, note that for a given partially-ordered plan, there are an exponentially increasing number of totally-ordered plans. For example, a partially-ordered plan for putting on socks and shoes that ignores the ordering of left/right foot first has six totally-ordered plans (called *linearizations*). A partially-ordered planner ignores ordering of actions if the ordering is not important to the outcome of the plan. Additionally, planners have found use for agents capable of executing multiple actions simultaneously, or creating plans for groups of agents acting cooperatively, so plans not enforcing a strict ordering of actions are desirable in those cases. Plans can also be combined with other plans at a later time, and a more flexible ordering of those plans may assist in their combination.

Partial-order planners begin by taking the initial state and goal states as the beginning and end of a plan respectively, then iteratively adding actions leading to intermediate states between the starting and ending states. At each step, the planner focuses its search by only adding actions that serve to meet a precondition of some part of the plan

that is not yet achieved. Additionally, the planner needs to check to make sure that the plan is *consistent* at each step; that is, the action added in a step cannot cause a contradiction in the plan. Contradictions can be formed by actions requiring an impossible ordering (A must be before B must be before A, etc.) or requiring a dual assignment to a variable (say, a variable v = A and v = B for $A \neq B$). Beyond these subtleties, this algorithm for a partial-order planner is both sound and complete, and forms the original POP planning algorithm.

4 Variations on planning algorithms

A great number of algorithms based on the same idea of partial-order planning have been developed, such as NONLIN, O-PLAN [5] and UCPOP [22]. However, several non-traditional approaches to planning have also been attempted with varying levels of success.

4.1 Satisfiability and planning

Many formalized analyses of planning algorithms approach the solution as a type of deductive reasoning. This method interprets the planning problem as finding a deductive proof in a system whose axioms state that action effects are implications of the action's occurrence when that action's preconditions hold. By formalizing planning in such a way, traditional methods of deductive proof can be used to analyze specific planning problems.

Deduction's complimentary problem is known as satisfiability [13], creating a model of a set of axioms. Kautz and Selman tested two different satisfiability algorithms, DP and GSAT, on a variety of planning-as-satisfiability problems, with several interesting results.

4.1.1 Differences between deduction and satisfiability

As mentioned earlier, deductive planning formalization is generally based on a logical system, such as the situational calculus or a specially-designed predicate logic system. Using this sort of language to represent a planning problem allows it to be solved deductively, by using the 'axioms' of action effects and preconditions. However, satisfiability moves in the opposite direction; starting with the 'axioms' of actions, it creates a model of the world that is consistent with those actions. This can easily lead to problems in which a plan satisfying the axioms is not a reasonable one. Put another way,

satisfiability can string together actions in such a way that the plan is consistent, but impossible to execute. For example, the formalization of an action as an axiom says that an action's effects take place when an action is performed while its preconditions are held true. This means that a satisfiability approach can execute actions without their preconditions being true, since it is consistent to say that the effect still occurs! (Recall in logic that the statement $false \rightarrow true$ is itself true.) This problem can be overcome by introducing the notion of time into the formal language, where one unit of time represents the time it takes for an action to be executed. Using this notion of time, actions can be rewritten to imply their effects at the time after they are completed, but also to imply their preconditions at the previous unit of time. In doing so, the satisfiability planner will no longer create situations where actions are performed without their preconditions being true. Other extensions to the planning-as-satisfiability approach include a restriction to only allow one action at any particular time, and an assertion that an action occurs at every time step. Since it is always possible to define a "do nothing" action, this last declaration is noted as not being particularly restrictive.

Satisfiability has the unique advantage of being able to place arbitrary constraints on the solution to the problem. Because of the introduced notion of time, it is easy to assert any particular attribute about the world should be held true at any particular time. Deductive reasoning systems have a much more difficult time enforcing such constraints. On the other hand, the explicit notion of time enforces a total-order-like structure to the resultant plan.

4.1.2 Summary of results

Several key elements contribute to the running time of the satisfiability planner. The length of the instantiated plan is bounded by $O(kc^d)$, where c is the number of constants (that is, the largest number of elements that could be assigned to a particular variable), d is the maximum depth of quantifier nesting, and k is the number of literals in the longest statement. Clearly, the depth of quantifiers in the statements has the greatest effect on the size of the output. Thus, replacing a predicate such as *move(x,y,z,i)* with three predicates *object(x,i)*, *source(y,i)*, and *dest(z,i)* has a greatly beneficial effect.

Better results were obtained when axioms that explicitly ruled out illegal states were included (such as an object being on top of itself). While plans leading to such states would fail, their explicit exclusion increased performance of the algorithm.

Two algorithms were used; one, GSAT, was a local greedy search algorithm, which performed well on general satisfiability problems, while the other, DP, a well-known backtracing algorithm, provided superior performance on a variety of traditional plan-

ning problems. While GSAT failed to solve several of the traditional planning problems presented to it on its first try, the exclusion of illegal states as mentioned above significantly improved its performance on those problems.

4.2 Planning with graph analysis

Another alternative way of looking at planning was proposed by Blum and Furst in 1997, using a structure they created known as a *Planning Graph* [1]. Their algorithmic planner, GRAPHPLAN, constructs a Planning Graph before searching for a relevant plan. Planning Graphs, which can be built in polynomial time and polynomial space, do not cover the entire search space of a problem, but rather represent an organized structure of possible actions for STRIPS-like domains.

GRAPHPLAN has several interesting features of note. First, like the planning-as-satisfiability planners above, it requires an explicit notion of time, where every action takes one time unit. This, again, does not restrict the plan generated to take a state-changing action at every time, so long as a "no-operation" style action is included in the allowable action set. Thus, GRAPHPLAN makes total-order planning-like commitments to the times in which actions occur. However, this is not to imply that the plans generated are total-order; rather, they imply orderings only among actions at different steps. For instance, GRAPHPLAN can generate plans where several actions occur at a particular timestep. Such plans therefore have partial-order flexibility concerning the ordering of actions during one arbitrary timestep. The authors also note that with multiple agents, or perhaps an agent capable of more than one action simultaneously, times in the plan where more than one action occurs could be handled in parallel by those agents.

GRAPHPLAN is provably sound and complete, and is also guaranteed to return the shortest possible plan for the given problem. It operates on traditional STRIPS-like problem domains, with the expected definitions of actions, objects, initial conditions and goal states. Additionally, it base algorithm has the attractive benefit of being extendable in order to detect if the goals of the problem are unattainable by any valid plan and halt with failure in such a case.

4.2.1 Planning graph structure

Planning Graphs are similar in structure to valid plans, meaning that each "layer" of the graph contains allowable actions for that given time. Actions and propositions alternate layers in the structure of the graph (see Figure 1). Unlike valid plans, actions at a given time in the graph are allowed to interfere with each other, meaning that actions at a

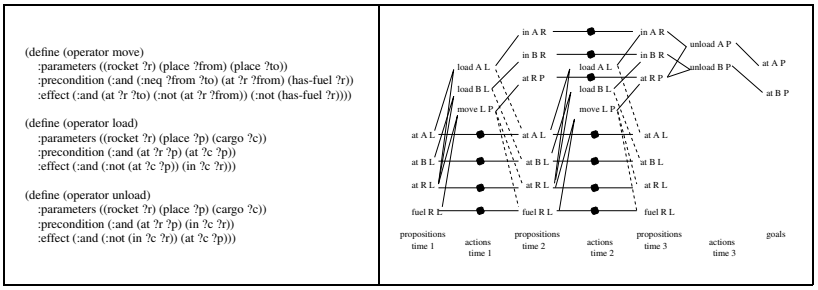

Figure 1: Operators and the generated Planning Graph for a particular planning problem. Taken from [3]. Dots in action layers represent no-op actions.

particular time in the graph could undo effects of each other. In fact, any particular level of the graph containing actions may contain every possible action such that the action's precondition are true at the previous level. This lack of restrictions on the graph make them relatively quick to create, and it is provable true that if a valid plan exists in n steps, such a plan is contained with a Planning Graph with n action levels (that is, a Planning Graph with n levels containing actions).

Once a Planning Graph is created, the planner examines it for actions that are mutually exclusive, which are actions at a given action level in the graph that no valid plan could possibly contain. While the planner does not find every mutually exclusive action pair, it does find a great many of them, which helps speed the search for a valid plan later. Mutually exclusive actions are identified by determining if they interfere with each other (one action deletes an effect of the other) or have competing needs (the preconditions of the actions are marked as being mutually exclusive at a previous time).

The planning algorithm itself is a backward-chaining recursive algorithm, taking advantage of actions marked as mutually exclusive in its previous steps. As before, the largest impact on the running time of the algorithm is the greatest number of formal parameters in the given allowable actions, so limiting the number of parameters within the operators of the problem is beneficial. Even with that note, though, GRAPHPLAN runs in polynomial time with respect to the number of objects in the problem, the length of the effect lists of the actions, and the number of initial conditions and operators. Of course, the required size of the graph (and thus, the size of the shortest valid plan) is a bounding factor as well.

4.2.2 Summary of results

GRAPHPLAN ran to successful completion in shorter CPU times (on slower computers, no less) than several competing planners, including PRODIGY and UCPOP. In general, the graph structure that GRAPHPLAN uses in its problem analysis seems to have provided it the greatest advantage over traditional planners.

4.2.3 Graphplan for multiple agents

The Duy Bui and Jamroga provided an extension for GRAPHPLAN for use in multi-agent environments [3]. In general, they provide for three classifications of multi-agent environments: agents that collaborate, working together towards common goals; agents that are adversarial and work against each other, or alternatively work towards opposing goals; and finally, agents that are indifferent towards other agents' goals.

A number of assumptions are adopted for this analysis. In particular, agents are assumed to have known goals and complete knowledge of their situation. Additionally, using the explicit time structure of the Planning Graph, agents are assumed to act in distinct turns. Finally, agents in adversarial cases adopt a kind of minimax planning goal; that is, they want to act such that they accomplish their goals, but will favor plans that protect them from the worst possible consequences.

The collaborative agent situation is not terribly different from a single agent acting towards a goal, so the GRAPHPLAN algorithm needs no real extension to handle collaborative agents. However, the authors add the concept of identifying the agent performing a particular action inside of the action instantiation, in order to ensure that a single agent does not perform more than one action simultaneously. Adding such a definition also allows the planner to provide an assurance that any particular agent only performs a single action at any point in time.

For the adversarial situation, both a forward-chaining and a backward-chaining algorithm are provided. While both algorithms provide plan searching capabilities for any given problem, their difference lies in the running time for a given type of problem. If the Planning Graph generated is a shallow graph, the forward-chaining algorithm provides superior results, even for particularly wide graphs (i.e., ones with high branching factors). Backwards-chaining, on the other hand, is a much better algorithm for graphs with a particularly high depth. It is conceivable, then, that in practice a planning system would first be well-served to examine the properties of the planning graph before searching for a suitable plan within it. In adversarial situations, the algorithms can be extended to provide plans accomplishing some partial subset of the desired goal state, should the complete goal state prove to be unattainable.

4.3 Applicability to games

Unfortunately, the differences in planning algorithms themselves are unlikely to produce significantly different results for artificial agents in games. While the algorithms themselves have interesting differences, at the end of their execution, a partial-order planner produces an applicable plan, which is all the agent (and the player) cares about. Indeed, the only specific instances of using planners in games uses a generic search algorithm, A*, over any other algorithm specifically oriented towards planning. While planning algorithms may work more efficiently on the problem space, the additional requirement on game planning to not adversely affect the framerate requires that the algorithm be fast and easily divided across several frames if necessary. It is possible that replacing A* with a planning algorithm could produce results faster, but it is more interesting to examine particular features of partial-order planners (such as scheduling) and try to apply them to an A* planner already implemented in games. It should further be remembered that the push for planning in gaming stems not from the desire to get results from any particular algorithm, but rather the ease of development of character behaviors and the sharing of those behaviors between different character types.

5 Parallelizing plans with partial-order planners

Many of the existing partial-order planners will return a plan in which actions could be executed in parallel, as alluded to above. Craig Knoblock provides an overview of the situations in which a general partial-order planner can be used to generate parallel-action plans, and further provides a classification of those situations, as well as an implementation of a partial-order planner that can generate such parallel plans [14].

5.1 When actions can be performed in parallel

Partial-order planners generally consider actions to be atomic, meaning that an individual action is completed uninterrupted without any influence from external factors. This is convenient when defining actions, since actions can specify their preconditions and effects without concern for other actions being executed simultaneously. Indeed, such definitions seem to preclude the notion of simultaneous action execution, as there is an unanswered question as to what conditions would need to hold for two actions to be executed simultaneously.

In parallel programming for multiple processors or threads, there are three possible kinds of conflicts: *procedural*, which is when an instruction must explicitly be ordered

before another; *operational*, when a resource needed may be unavailable; and *data*, when one instruction requires the result of another. Data conflicts are analogous to actions accomplishing preconditions of other actions, and procedural conflicts are already handled by partial-order planners by their explicit order of (some) actions. Thus, the only type of conflict not explicitly handled by partial-order planners is a resource conflict. Several planners generate parallel execution plans by simply ignoring the problem, assuming all actions to be independent from each other. However, planners can be extended to detect resource conflicts and correctly plan for them with the simple addition of an explicit representation of resources.

5.2 Types of parallel plans

Knoblock identifies four different types of parallel plans that may be generated: plans with *independent actions*, plans with *independent actions relative to a goal*, plans with *independent subplans relative to a goal*, and plans with fully *interacting actions*.

5.2.1 Independent actions

Independent actions are the simplest type of parallel execution plans. Such plans enforce that any two actions executed in parallel be completely independent of each other. Here, *independence* is defined as the effect of the actions being executed in parallel being the same as "...the union of the effects of the actions being done in isolation." [14] In other words, the effect of the actions in parallel must be the same as the effects of each action done in in sequence, or that the effects of the actions don't interact with each other. It should be noted that partial-order planners that generate unordered actions does not necessarily imply that those actions are independent.

5.2.2 Independent actions relative to a goal

Several planners, such as UCPOP, only allow actions to remain unordered when there is no *threat* between those actions. Two actions are said to threaten each other when one action could delete a relevant condition with respect to the final goal state. Conditions, then, are defined to be relevant when they are either a condition of the goal or a precondition of an action that achieves a relevant condition.

5.2.3 Independent subplans relative to a goal

Once a planner determines all relevant conditions, it could then examine threats not only at an action level, but on the level of subplans, possibly producing several actions

in a row that can be executed in parallel. Subplans, then, are independent relative to a goal if, for all conditions relevant for that goal, executing the subplans in either order has the same effect as executing the subplans simultaneously.

5.2.4 Interacting actions

Interacting actions are ones that alter the total outcome based on their simultaneous execution. Specifically, if the effect of executing the actions simultaneously is different than the combined effect of executing the actions one at a time, the actions are said to be interacting. In order to account for the possibility of interacting actions, a planner needs either an explicit or an implicit representation of time, and most partial-order planners do not. In the next section, a method of dealing with concurrent interacting actions will be presented.

5.3 Partial-order planning with interacting actions

Temporal planners are planners that deal with an explicit representation of time when forming plans, and while these planners are certainly helpful when dealing with problems requiring interacting actions, in many situations, they may not be necessary. Boutilier and Brafman present a simple extension to STRIPS-style problem representations that allow for parallel execution plans with interacting actions [2]. Specifically, the algorithm they present is targeted towards an agent that can perform multiple actions simultaneously, although this is equivalent to a situation with multiple agents that can act independent of each other.

Actions can interact in several different ways; their interactions can be negative, such as when one action cancels the effect of another, or they may be positively interacting, such as when an intended effect is only achieved when two actions are performed simultaneously. Therefore, a planner needs to be able to specify which actions occur at some particular time. More generally, the planner should be able to specify that certain actions must or must not be performed at the same time as another action's execution. One way of doing this is to treat every possible combination of available actions at a given time as one action. For example, say we have two agents, A_1 and A_2, both of which can perform actions b, c, d. At any given time, the planner could specify A_1 doing any of those three actions, as well as A_2 doing any of the three actions. If we represent these as an ordered pair, our possible actions are $(b, b), (b, c), (b, d)$ and so on, for every possible combination of action for each agent. However, this way of representing joint actions increases exponentially in size as the number of agents and

actions increase, and further enforces all agents to be performing a specific action at any time that one agent is performing some action.

Boutilier and Brafman therefore propose the addition of a *concurrent action list*, a description of the actions that can and cannot be performed simultaneously with a given action. Of course, the action descriptions themselves must also be modified so that the planner can determine the differing effects of the actions when they are performed in conjunction with some other action. This is done by introducing a *when* clause to the effect list. The *when* clause specifies that if an action is performed while some other event is true, the effect of this action is modified to include the antecedent of the *when* clause. This allows the planner to determine how the outcome of specific actions will change depending on other simultaneously performed actions. Actions can have zero, one, or many *when* clauses, with the restriction that if multiple *when* clauses exist, the preconditions of those clauses must be disjoint.

Additionally, actions have been extended to include variables specifying the agent performing the action. This allows the planner to ensure that no agent is performing more than one action at a time, as well as that no action is being done by more than one agent concurrently.

With these definitions, a joint action (several actions performed simultaneously by several agents) is defined to be consistent if $a)$ all of the actions' preconditions and *when* clause preconditions are not contradictory, $b)$ all of the actions' effects are not contradictory, and $c)$ the concurrent action list of each action is satisfied for the rest of the actions within the joint action. With this, we can specify valid joint actions, and develop a partial-order planner to generate joint actions for multiple agent situations.

5.4 Applicability to games

The analysis of interacting actions has several important implications for games. First, examining interacting actions has immediate applications for a single agent merging plans. In any situation where an agent could create plans to accomplish more than one goal simultaneously, the agent must examine the ways in which the goals and actions contained within the generated plans interact. Furthermore, agents acting simultaneously to other agents could clearly benefit from coordinating their actions to achieve maximum results, or alternatively restrict or thwart the results of an opposing player.

6 Plan recognition

A closely related field to planning is *plan recognition*, where agents attempt to determine the plans and goals of other agents through their observations of the world. Mao and Gratch present an overview of plan recognition and discuss some approaches to the problem [15].

6.1 Plan recognition overview

Plan recognition refers to the problem of determining the plan of other agents through the observation of their behaviors in the world. An agent attempting plan recognition observes a sequences of actions in some other agent, then attempts to map those actions into some sort of plan representative of the goals of the other agent. One of the chief problems in plan recognition, then, is disambiguating among possible plans that match the observed sequence of actions.

There are three main categories for types of plan recognition. Cohen et al identify *keyhole* and *intended* recognition [4], while Geib and Goldman deal with *adversarial* plan recognition [9]. Intended recognition describes the situation in which the actor whose plan is to be determined is cognizant of the intended recognition and performs actions in order to facilitate the plan determination. Conversely, in adversarial recognition, the agent is aware of the intended recognition but takes actions to deliberately hinder recognition. Keyhole recognition is the most common form, where the observed agent makes no attempt to influence the recognition process in any way.

The actions that the observer is able to see also lends itself to a classification: if the observer detects every action in the subject agent's plan, this is known as a *fully-observable* action sequence; otherwise, the action sequence is *partially-observable*. Partially-observable systems include situations where some actions are unobserved, or cases where actions themselves are unobservable but may have some observable effects.

6.2 Planning and plan recognition

In some sense, plan recognition is the inverse problem of planning. Partial-order planners take a goal and generate a plan intended to accomplish that goal, whereas plan recognition finds the intended goal or plan of an agent when presented with a list of possible plans. Therefore, a plan recognition system could conceivably use a planning system to generate a list or repository of possible plans.

6.3 Applicability to games

Plan recognition could be used by agents working with or against a human player. An artificial agent that recognizes the goal of a player based on the actions the player has taken can then take steps to assist or hinder the player in whatever endeavors are being pursued. Clearly, an agent cannot take these steps without explicitly knowing what is to be accomplished.

7 Plan merging

While many partial-order planners specialize in creating a plan for a single goal, this is rarely representative of intelligent agents in nature. Typically, rational agents have several goals to pursue simultaneously. While it may be the case that an agent may choose to concentrate its efforts on a single goal, it would be overly restrictive to enforce that our agents may only pursue one goal at a time.

There are additional considerations when attempting to plan for multiple goals, of course. It would not appear rational if an agent were to attempt to pursue two conflicting goals simultaneously, for example. Furthermore, if two or more goals have positive influence on each other, it would behoove the agent to take advantage of that overlap. If our agent were to make a plan to obtain a screwdriver and a hammer, it would not appear rational for the agent to go to the hardware store, buy a screwdriver, return home, then go back to the hardware store, and finally buy a hammer. Certainly it would appear more intelligent for the agent to go to the hardware store once and obtain both items before returning home. We would like our agent to recognize the overlapping parts of the plans and take advantage of them, should they exist.

7.1 Positive goal interactions

One way of identifying possible positive goal and action interactions involves examining the subgoals of generated plans. In the system described in [25], agents have pre-defined plans used to achieve their goals. Subgoals are defined as the conditions necessary for successfully completing a plan. In a sense, subgoals are the preconditions of the actions in a given plan. From this, agents can create a *goal-plan tree*, consisting at alternating levels of goals and plans. The top level goals are high-level goals, decided upon by the agent. The next level down, then, would be plans that could be used to achieve those goals. Each plan would have sub-goal nodes below them, representing the conditions needed for that plan to be completed. In turn, those sub-goals would

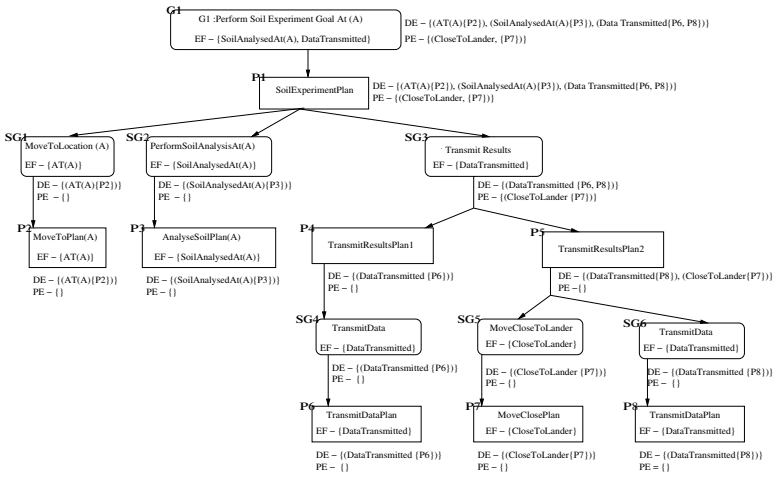

Figure 2: An example goal-plan tree. From [25].

EF – Effects DE – Definite Effect PE – Potential Effects

DE – { (EF, [plan, plan ..]), (EF, [..]) ..} each effect has the set of plans that bring about the effect. Similar for PE .

SE (G) – Effect–summary of goal G. It is a two tuple < DE, PE >

DMP – [(EF, goal{plan, plan}, goal{plan, plan}) , (EF , goal{..}, ..)] is the data structure that contains plans that can be definitely merged. With each effect the goals and the plans of the goals that bring about the effect is stored.

PMP – [(EF, goal d/p {plan, plan}, goal d/p {plan, plan}) , ..)] similar to DMP except that the plans here are only potentialy mergable. For each goal of an effect the flag (d or p) indicates if the effect is definite or potential.

Common effect of G1 and G2 – an effect that is an effect of G1 and an effect of G2

WGL – The Waiting Goals List. A list of goals currently suspended due to scheduling. Assists in deadlock prevention.

Figure 3: Symbols and terms used in Figure 2. From [25].

have plans that could accomplish them, and so on. An example goal-plan tree is shown in Figure 2.

An interesting note about the proposed system is the pre-generation of plans. Each agent has a set of goals and actions available to it, and so it is possible to develop a comprehensive list of all possible plans an agent could execute. One consequence of this is that it become easier to match plans to goals at runtime, meaning that the system need only look up all applicable plans for a particular goal. If a plan fails, the agent can mark that plan as being no longer applicable and move on to another plan. This also makes it easy to identify cases where the agent has no other available plans to try. A direct consequence of this is that individual goal instances need to keep a list of the available plans that can satisfy them.

Another difference in this system as compared to other partial-order planners is the introduction of *in-conditions*. In addition to plans and goals knowing of their possible

pre-conditions and effects, in-conditions are a way of representing a condition that must be true throughout the duration of the plan or goal. For plans and actions, it is simple enough to treat in-conditions as a pre-condition, since a plan that begins executing without satisfying an in-condition will immediately fail.

For planning, an agent initially decides upon the goals it wishes to pursue. This creates a goal-plan tree with all possible plans that can be used to achieve the agent's goals. From there, the planning system can examine the goal-plan tree to determine the effects of each plan, and classify them into *definite effects* and *potential effects*. Definite effects are those effects that must always take place for a given goal or plan. These effects are either those that are required by a goal or every subgoal, or side effects of particular actions that cannot be avoided, no matter what plan is chosen. Definite effects are not necessarily required by every plan, but every definite effect is required by at least one plan in every possible way the agent has of accomplishing its goal. Potential effects are those that only occur down at least one plan path, but not all paths. Potential effects can occur with many different plans, but do not occur for all plans. There are two further important notes about potential and definite effects: first, the sets are defined to be exclusive, so any effect that is a definite effect is by definition not a potential effect; second, all effects hold only if the plan's execution is successful.

Potential and definite effects tell us precisely how the goals and actions in our goal-plan tree interact. Thus, every time a plan is successfully completed or fails to execute to completion, the tree must be updated to reflect the changes in possible and definite interactions. This same logic holds for sub-plans, as well. With this information, though, we can build data structures to inform the planner of plans that could benefit from being merged. *Definitely mergeable plans* are plans that share some definite effect. *Possibly mergeable plans* are all plans that could possibly be merged based on some shared effect.

Definitely mergeable plans necessarily share definite effects, but potentially mergeable plans could have any effect in common; the effect could be a definite effect of one plan and potential effects of the rest, potential effects of all plans, or some combination of definite and potential effects of the plans in question. Agents have the choice to be cautious about potential effects and always execute all plans concerned, or optimistic about an overlap in plans and effects by only executing some part of the plans involved. Depending on whether the agent is more constrained by time or resources, one of the two approaches may be clearly better for the agent. In any case, an agent can examine several plans to determine their effects and use their interacting effects to decide on how to merge the plans. For any two plans with the same single effect, the agent can pick to execute one plan or the other. If one plan achieves a potential effect that the

other does not, the agent can again pick to execute either plan, since potential effects are not required for the goal. If one plan achieves a definite effect the other does not, the first plan must be executed for the successful completion of the goal. Finally, if the plans have differing definite effects, the plans cannot be merged and both must be executed for the successful completion of the goal.

In order to accomplish maximum utility out of plan merging, it may be necessary for the agent to delay the execution of some plans until other plans are ready to execute. This, in particular, is where an agent being cautious or optimistic about potentially mergeable plans comes into play. An overly cautious agent may end up not merging plans that could have been, resulting in wasted and unnecessary effort. On the other hand, an overly optimistic agent could wait to merge a plan that has no chance of being merged! This is known as *useless wait*, and is extremely inefficient. However, it is more unlikely to occur with plans that have only a single effect, and can be either eliminated or reduced through goal filtering. Balancing the goal filtering is important to get the maximum efficiency out of an agent, as strict goal filtering can have the effect of making a large number of mergeable plans run individually. Less strict strategies, though, have been relatively effective in reducing useless wait, although these less strict methods cannot guarantee that useless wait will never occur.

7.2 Avoiding resource conflicts

The opposite problem from taking advantage of positive goal interactions is avoiding conflicts between goals or actions. The appearance of rationality in an agent can be severely hampered by the agent attempting to pursue goals that conflict in any way, so agents and planners must take care to identify any potential conflict and prevent it from becoming an issue. Some conflicts are easily identified, such as when goals have clearly conflicting world states, or when one action undoes a precondition or effect of another action. Thangarajah et al present some issues surrounding conflicts of resources [26].

Resources, like effects above, can be classified as being necessary, meaning that the resource is required for every possible way of accomplishing a particular goal, or possible, for resources that may be used in some plans for a goal but not every possible plan. Resources may also be consumable and lost forever once used, or they could be reusable and available for use again after a particular action or agent is finished with it. An agent that considers resource conflicts in planning to avoid irrationally pursuing multiple plans using resources must be able to reason about these different types of resources. For example, if an agent has 10 units of energy and two goals, each requiring 5 and 8 units of energy respectively, it is certainly irrational for the agent to

pursue both goals, as one or the other will certainly fail. On the other hand, an agent with two goals requiring a single reusable resource may be able to accomplish both goals with some scheduling.

Again, a system is proposed involving complete foreknowledge of all possible plans for all possible goals, allowing an agent to quickly determine every possible path available to it in order to accomplish its goals. In order to allow agents to reason about resources, a list of every possible resource is defined. Then resource requirements on actions and plans can be formalized into a list of resource types with the amounts needed. For each action defined, a list of all required resources is also defined such that every resource type is listed once in the definition, with an amount of zero for resources not required. Then for entire subgoals, an agent can compute the total amount of each resource that is necessarily required or possibly required. Unlike potential and necessary effects, any necessarily required resource is by definition possibly required as well. In this way, the set of all necessarily required resources for a plan forms the lower bound of resources required, while the set of possibly required resources gives an upper bound on resource amounts.

From these definitions, agents can again combine necessary and possible resource amounts for individual actions or entire subplans to determine resource requirements when those actions or subplans are combined. Furthermore, the agents can examine the different resource requirements for when the actions or subplans are to be performed in sequence or in parallel. With this information, an agent can determine that a given plan will certainly fail if its necessary resource requirements are above the available resource amounts, or that a given plan can always succeed, no matter what order the actions are performed in, if its combined possible resource requirements are less than the available amounts. In cases between these two, it is not clear if the agent will be able to accomplish all desired sub-goals. There are two sub-cases here for agents being able to determine more information about the goals. Goals are said to be *schedulable* if the planner can guarantee that there will be sufficient resources available to accomplish the goals provided that the plans are executed in the proper order. Goals are *schedule-dependent* if the planner can determine that if resources are not properly used, then some part of the plan will fail due to lack of resource availability. Goals can be schedulable and schedule-dependent simultaneously.

7.2.1 Applicability to games

On an individual basis, examining the interactions of actions between several plans allows an agent to plan more effectively for several prioritized goals simultaneously.

Especially for games where an agent has a wide variety of goals and many ways to accomplish them, careful examination of the varying plans could quite easily lead to a more intelligent opponent for a player. Within a group, agents examining interacting actions would more easily be able to coordinate actions and produce more realistic group behavior.

7.3 Operator merging

Another approach to plan merging takes the merging operations down to a lower level, concerning itself with the merging of operators (actions) rather than entire plans. Foulser et al [8] present a series of plan merging algorithms concerning themselves with grouping actions together based on their effects and their relations. In general, two or more actions can be grouped together if there exist no other actions outside of the group that would have to occur in between the execution of actions within the group. An example of this would be a chain of three actions, each establishing preconditions of the next action in the chain. In this case, the first and third actions could not be grouped individually, since the second action must occur between the execution of the first and third actions. However, the three actions may be grouped as a whole.

Grouped actions can then have their net preconditions and useful effects defined. Here, net preconditions is simply the logical set of preconditions of actions within the group that is not achieved by actions within the group, and useful effects are those effects of the actions in the group that establish preconditions for either other actions in the plan, or establish part of the goal directly.

Grouped actions between plans can finally be examined for replacement by a single action with the same set of useful effects. Ideally, the cost of this replacement action should be less than that of all the grouped actions.

7.3.1 Applicability to games

As this algorithmic approach does not require a large number of plans, it is more readily applicable to time-sensitive situations that often occur in real-time games. An agent could generate a plan for its most important goal, then generate a plan for its second most important goal, and examine just the two plans for useful operator merging situations. Since actions that have different useful effects cannot be merged, an agent with contradictory goals will end up with two completely separate plans that must be executed independently.

The algorithms presented have running times dependent on both the length of the plans presented and the number of different actions contained within the plan. In fast-

paced games such as first-person shooters, plans should be relatively short as agents tend to not have much time to perform complicated actions and maneuvers, keeping the execution time of these algorithms down to a reasonable amount.

8 Hierarchical Task Network planning

Hierarchical Task Network planning was developed shortly after STRIPS-style planners, although much more analytical work has been done on STRIPS-style partial order planners. More recently, though, HTN planning has been garnering more attention in academic literature and research. Erol et al present an overview and analytical semantics for HTN planners in [7] and [6].

8.1 Overview of HTN planning

HTN planners share some similar ideas with partial-order planners. Each represent world states with a collection of atomic statements, and actions change the world state through associated effects. The most important difference is how HTN planners represent their desired world state changes. In partial-order planning, the planner examines a goal state, then comes up with a series of actions designed to change the world in such a way that the desired goal is met. Instead of goals, HTN planners use *tasks* and *task networks* to bring about specific changes in the world. Tasks are divided into three main categories. *Goal tasks* are specific properties in the world that the planner attempts to make true by the end of the plan. These can generally be represented as a conjunction of literal statements, just as partial-order planning goals are. An example of a goal task would be a state representative of "owning a house." *Primitive tasks* are atomic tasks that can be accomplished by an agent with a single action. Any task that corresponds to a partial-order planning action can be considered a primitive task. Some examples of primitive tasks are tasks such as buying lumber or nailing two boards together. Finally, *compound tasks*, or non-primitive tasks, are higher-level tasks that cannot be represented atomically. Compound tasks can be thought of as any desired change that can be constructed out of goal tasks and/or primitive tasks, or anything that can be accomplished in a variety of different ways. "Building a house" is a compound task, since constructing a house is made up of many smaller tasks.

In HTN planning, tasks are strung together in structures known as *task networks*. Task networks are ways of organizing tasks together to form a unit with some associated effects. These task networks are analogous to actions or even subplans in partial-order planners. During the planning stage, compound tasks are replaced with

1. Input a planning problem **P.**
2. If **P** contains only primitive tasks, then
   resolve the conflicts in **P** and return the result.
   If the conflicts cannot be resolved, return failure.
3. Choose a non-primitive task t in **P.**
4. Choose an expansion for t.
5. Replace t with the expansion.
6. Use critics to find the interactions among the tasks in **P**,
   and suggest ways to handle them.
7. Apply one of the ways suggested in step 6.
8. Go to step 2.

Figure 4: Basic HTN planning algorithm. From [7].

task networks based on applicability. Each task network that accomplishes a given compound task is denoted as such in a *method*, indicating that the compound task can be replaced in the plan by the given task network. Note that task networks themselves are not restricted on the type of tasks they can contain, allowing task networks themselves to have multiple compound tasks if necessary. Additionally, task networks can be associated with any number of methods, and in turn can be accomplished by any number of different task networks themselves. Owning a house, for example, can be accomplished by building one, buying one, or winning a contest.

The basic HTN planning algorithm is shown in Figure 4. Steps 3–5 are the expansions steps, in which the planner replaces higher-level compound tasks with task networks. However, the plan is not guaranteed to be free of conflicts at the end of step 5. Step 6, then, is an opportunity for the planner to deal with any kind of conflict present in the plan. Traditional conflicts, such as deleted preconditions, are dealt with here, but any kind of conflict could be included. Critics are a way of identifying potentially crippling interactions in a plan early, in order to limit expensive plan backtracking. [24] provides an overview of the various ways critics have been used in HTN planning.

Critics contribute to one of the major benefits of HTN planning, especially when being applied to agents that act in real-time. If a plan fails, generally it will mean that a particular action or task has failed to be completed successfully. In partial-order planners, there is no formal mechanism for planning from a partially complete plan. HTN planners, however, have the option of just replacing the failed task network with another based on alternative methods [27]. If no other methods remain, the planner can

replan in a more traditional way.

It should be made clear that while the output plan from HTN planners is made up of the same kind of atomic actions that partial-order planners produce, there are substantial differences in the kinds of problems that HTN planners can solve. While both partial-order planners and HTN planners could derive a plan to bring an agent to a different city, HTN planners have the ability to create a plan for a round-trip vacation. Traditional partial-order planners would have difficulty even expressing this as a goal, since the desired end state is the same as the starting state! Indeed, it is provably true that HTN planning is more expressive than traditional planning, which does not include any kind of decomposition.

8.2 Applicability to games

The ability to quickly replan specific parts of an otherwise valid plan is important for agents in a quickly-changing environment. In a game environment, it would save valuable computation time for an agent to just discard a part of a plan, rather than have to plan again from scratch. Agents with similar or the same task network within their plans may be able to use the matching task network to more easily coordinate their actions, producing better squad-based behaviors.

9 Applications of planning in games

The core of any artificial agent is a decision-making process. Whether the agent is a research robot doing scientific testing or a computer chess opponent, the only real output an agent has are the things that it does as determined by its own internal decisions. Therefore, the only qualitative differences between two artificial agents are the decisions that it makes.

In many real-time simulations, agents use either finite state machines or rules-based systems to make decisions as to what actions they take. Finite state machines keep track of an internal state that determines an agent's actions. Based on some internal mechanism or an external force or observation, the rules of the state machine determine what state an agent changes to, thus changing the behavior and actions of the agent. Rules-based systems match external stimuli to a series of rules the agent has for determining behaviors. The best-matching rule for a given situation determines the action an agent will take.

While finite state machines and rules-bases systems are widespread, they are not the only way of controlling an agent's decisions. In fact, as worlds and desired behaviors

become more complex, so do the complexity of state machines and the number of required rules. An agent's decision-making processes need not be one of these two systems, though; so long as we can represent the world state and an agent's goals and actions, we could use a planning system instead.

Jeff Orkin presents several key elements to a real-time planning system for use in games in [18] and [17]. The first is that a goal planning system composed of small, atomic actions without an explicit coupling to the goals they might accomplish allow a natural division of work between engineers and designers. Designers can focus on what actions an agent is able to perform, and let engineers worry about the details of how and when the agents perform actions. While it is certainly possible to design rules-based systems of finite state machines that are data-driven, such systems are still heavily technical and require an understanding of the workings of FSM's and rule-based systems. This is not an ideal system, as designers are not always able to concern themselves with the inner workings of particular algorithms.

Another advantage of goal-based planning are that complex behaviors occur naturally due to the nature of partial-order planners. An agent that comes across an obstacle to completing its plan can simply replan to find another solution to its problem. Finite state machines or rules based systems could handle the same situation with the same results, but they would need to have explicit rules for every conceivable kind of obstacle, whether it be a grenade or simply a blocked door. With goal planning systems, these behaviors come out of planning naturally, with no need to write specific rules.

The major challenge for implementing a goal planning system for games is implementing a symbolic representation of the world. Agents need to be designed in such a way that they have an internal world state, representing their interpretation of the world space [19]. These can be represented as facts, each containing a value and how confident the agent is in their correctness. Goals, then, can be represented as a target world state based on some combination of target values. Actions store their effects upon the world, and can be stored in a hash table on those values in order to be quickly found by the planner. Additionally, actions can specify *context preconditions*, or preconditions that the planner should not try to make true through other actions. For example, if the "GetToCover" action requires a valid and available cover position to be near, there is nothing an agent can directly do to create a cover node if one does not exist. A context precondition can inform the planner that this particular action is not currently available, based on the current world state of the agent.

Rather than use an existing partial-order planning algorithm, it is possible to create plans using a traditional search algorithm, A*. A* was chosen for a variety of reasons [20]. First, most games already have a highly-optimized, generic A* search

implemented, so reusing an existing algorithm saves time and effort in both development and testing. A* searches are also easy to divide across several frames, which is a necessary requirement for games; AI planning must not adversely affect the game's framerate.

10 Proposal for research

Based on the research presented, the most directly applicable area for improvement for planners in games seems to be the addition of analysis into interacting actions for both individual agents pursuing multiple goals and multiple agents working in a group. I would like to produce a planning system based on Jeff Orkin's A* planner and implement several improvements, including scheduling and ways for an agent to generate plans to pursue multiple goals simultaneously with respect for interacting actions between merged plans.

Once this functionality is in place, the planner could be extended to account for higher-level goals on a squad level for a multitude of agents working together. This sort of higher-level planning would allow greater cohesion between cooperative agents working towards a common squad goal. The interactions of this higher-level "squad" planner and each agent's individual planner would be have to be specified, allowing the agent to obey orders but override orders for more pressing individual needs. The specific behaviors resulting from this planner should be quantified and differentiated from the results of a squad of agents acting only with individual planners.

Besides scheduling, defining compound actions could be a useful addition for the designer. Should a designer desire that a specific chain of actions should occur, such behavior should be allowable within the language of the planner.

The output of this research would be a fully functional A* planner suitable for use in real-time gaming applications. The additional features above and beyond those already used in games would be examined and analyzed to show specific advantages or disadvantages over the simpler planner, with respect to running time, language complexity for specifying goals and actions, perceived intelligence of the behaviors of the agent, difficulty of creating complex and realistic behaviors of agents, and workflow considerations for both engineers and designers in using such a system.

11 Research products

As proposed, a full planning system suitable for use in games was produced. The planning system is functional and was used to control the AI opponents in a graduate-team flight combat game, *Paper Cuts*, produced for DigiPen's GAM550 and GAM551 classes.

11.1 Planning system

The planning system is described in detail in the following sections. Particular components of the system are described in their own section for convenience.

11.1.1 Search algorithm

After careful consideration of the various algorithms, it was decided that the A* search algorithm was best suited for use in the searching part of the planner. Following the reasoning of the discussion above, there were several reasons for choosing A* over a more particular planning algorithm. First, it should be noted that A* is perfectly suited for planning, as planning ultimately comes down to a guided search with heuristics. Second, games are unique in having a requirement that any planning done must not affect the framerate of the game. Ideally, the plan processing should be split up over several frames of the game if the algorithm requires any significant amount of time to create the plan. A* is easily split up over several frames of running time. Third, A* is a well-examined algorithm with many proposed speed improvements and widely accepted strategies for writing a generic algorithm that can nonetheless be tuned for improvements in specific cases.

The A* search algorithm implemented takes into consideration the advice offered in a variety of articles, in particular the strategies for making the algorithm generic and fast given in [10] and [11]. Without using templated classes suggested in [10], the A* algorithm uses hierarchically defined Goal and Storage classes for representing whatever states are necessary to search and store. In this way, different Goal classes can be defined to specify target states, and different Storage classes can be defined for particular needs in storing and retrieving search nodes off of the open and closed lists used in the A* algorithm.

A Node class is defined to handle the specific needs of keeping track of a node's cost, its parent and children nodes, and whether a search space is already on the open or closed list. Storage classes are, at simplest, a organized collection of Nodes. These storage classes can be customized for the particular needs of a specific type of Node.

For the purposes of the planning system, a Storage class was implemented to store nodes on the Open list in a priority queue structure, such that the cheapest possible node was stored for quick retrieval. Other storage classes are suggested in [11], but were not implemented for this planner. If the A* algorithm was to be used for pathfinding as well as planning, alternative storage classes might be prudent to develop and use, but this was not necessary in the course of development for either the planning system or *Paper Cuts*. Likewise to the Node and Storage classes, different Goal classes can be defined for particular needs from the A* algorithm. Again, neither the planner nor the game needed these additional class definitions, but having a generic Goal class that can determine which child states are valid for the goal and when the goal is reached was a useful design for creating a variety of different goal states for agents in the game.

11.1.2 Goal classes

Goal classes are defined to have a name, a priority, and list of target world states. In this way, Goals themselves can determine whether or not they have been accomplished and when it is time for them to run. The mechanism for determining what goal to run is described below, as is the definition of target states. For now, it is enough for the description of the design to note that these desired target states allow the Goal to be able to determine if the current state of the world is such that the goal is accomplished. Additionally, Goals can determine if taking a particular action would bring the world state closer or further away from accomplishing the goal. Towards this end, the Goal class has functions taking a single action and returning values indicating if this action would help accomplish any part of the goal. A Goal can perform similar functions on an entire plan, with a function defined to quickly tell if the goal is complete or not after the plan is completed, and if not, what properties of the world remain to be accomplished after running the plan.

Outside of functionality for searching, goals have a function to produce a single goal from two disparate goal types, and a list of goals that this particular goal is exclusive with. These aspects of goals are discussed in greater detail below in the section on goal merging.

11.1.3 Action classes

Actions are descriptive of how the world simulation changes in response to agents completing the action, but in the context of the planning search, it is more instructive to think of them as the nodes over which A* searches. In this sense, Action classes are simple classes containing world state preconditions for running and world state effects

after the action is complete. As described in [17], Action classes also have *context preconditions*, which are any kind of preconditions that need to be true for the action to be successfully completed, but require more intensive computation to determine or do not fit neatly into the particular world state definition.

Besides the list of preconditions and effects used for search, an Action class can define a precedence list of actions it should run before. This is used to schedule actions and is described in more detail below.

Most importantly, Actions define logic for an agent to use when performing the action. For example, a FireMissile action would define logic for an agent to obtain missile lock-on and perform the firing action. Actions return a value to indicate when they are complete or have failed. In this way, an agent knows to move on to the next action or report the plan as failed.

Originally, Actions were implemented following the Singleton design pattern, as an individual Action class describes a specific and unchanging outcome and action. However, as improvements to the planning system were made, this was changed so that each agent would own an instance of each particular action it could undertake, and that copies of these action instantiations could be made as necessary. This became necessary as particular versions of plan merging were implemented in the system.

11.1.4 Plan class

The Plan class is merely a list of Actions to be performed. This is the ultimate output of our planning system, and the structure used by the agent when actually executing actions. Plans were very simple, containing only this list of actions and a pointer to the current action in the plan being performed. This kept the agent from having to know how far along in the plan it was, while still providing a convenient means of determining when a plan was complete.

11.2 World representation

Actions need to have a way of representing the effects they have on the world and any requirements of the world state that must be met before the action can be run. Likewise, agents need a way of representing the world state as they detect things in the world, learn new facts, and forget old ones. This current state is the starting point for the search when agents are determining their plan.

The main structure through which agents keep track of things in the world is WorldState, which is merely a list of WorldProperties. WorldProperties, in turn, are simple

key/value structures representing things about the world that can be changed via some action. For example, the keys used in *Paper Cuts* were

- `kTargetIsDead`,
- `kTargetInMissileRange`,
- `kTargetInGunRange`,
- `kAtLocation`,
- `kPatrolling`,
- `kHasItem`,
- `kReturnedItem`,
- `kBelowMinimumAltitude`,
- `kProvideCover`,
- `kDie`

Each WorldProperty has an associated value, which can be a boolean, integer, or floating-point value. Additionally, WorldProperties have space to keep track of an object id, used to refer to an important reference object. For example, a WorldProperty with the `kProvideCover` would keep an object id of the game object that this agent is supposed to be providing cover for. In this way, agents are able to keep track of the current state of the world as dependent upon their actions, and thus be able to provide the planning system with their current state and the actions they can use to change that state.

Agents need a different way to detect and keep track of things in the world outside of their direct influence, however. Enemy agents need a way of tracking the player and representing that location. The precise methods of agent sensing are outside the scope of this system description, but the storage of these facts can influence the values in WorldProperties. Agents store facts that they become aware of in WorkingMemoryFacts, which are modeled after those described in [19]. WorkingMemoryFacts have a templated type for whatever value needs to be stored and a float value indicating the confidence in that value. For example, when an agent detects the player on radar, it stores the vector of the player's position in a WorkingMemoryFact with a confidence value of 1.0f. As time passes without the agent seeing or otherwise detecting the player, this confidence value decays down to 0.

11.3 Agent definition

Agents are the computer-controlled characters in the world, also known as non-player characters or NPC's. All agents have a memory system, a sensory system, and goals and actions as discussed above. Other information may be stored on the agent depending on the needs of the simulation, such as an agent's location or inventory, but these details are at most used to check particular preconditions or context preconditions of actions during planning and are otherwise unimportant to the discussion here.

In the system created, agents are loaded into the game with a particular "type," with the type defining an agent's goals and actitons. The agents to be loaded and their type are defined in a Lua scripting language file, which is loaded when the game starts running. Which goals and actions are given to a particular type of agent is defined within C++ code. Once the game is running, though, agents are unaware of their type and the processing of all agents is handled in the same way. The behavior of different agents then comes from the differences in their allowable goals and actions and the relative weighting or importance they give to particular goals at particular times.

11.4 The Planning Algorithm as used in an Agent

In the traditional Sense-Think-Act cycle for artificial agents, our planning algorithm only takes care of the Think part of the cycle. Thus, first an agent must complete a sensory update of the world. If this sensory update indicates that the agent should replan, or if the agent doesn't have a plan to begin with, the agent potentially needs to come up with a new plan. The first step in determining if a new plan is needed is to reevaluate which goal is most important to the agent at the current time.

In the system created, all agents own copies of the goals that they can pursue. The relevance of the goal is stored as a floating-point number on the goal itself. Events in the world, such as noticing an enemy or taking damage, change the importance of particular goals as necessary. Once the most important goal is determined, the agent attempts to make a plan for that goal. However, it's possible that the agent is unable to create a plan to accomplish that goal, in which case the goal needs to be marked as impossible to achieve. This marking persists until the world changes in such a way as to make it possible to accomplish that goal, at which point it can be reevaluated during a regular update. One final caveat exists: if the agent's most important goal is the same as the goal for which its current plan was made, the agent shouldn't replan for the same goal and should instead just continue executing its current plan. Figure 5 shows the code used to determine the best goal and create a plan for it.

This system as described is a fully implemented planning system, suitable for use

```
bool newGoal = false;
if(m_shouldReplan || !HasCurrentPlan()) {
    //Check to see which goal is most important to the agent right now.
    // If the most important goal hasn't changed, don't bother to replan.
    // Note this means if a plan fails, we'll need some way of ensuring the agent doesn't continually attempt the same
    // (failing) plan
    bool hasPlan = false;
    do {
        Goal* agentGoal = GetBestGoal();
        if(agentGoal != GetLastGoal() || !HasCurrentPlan()) {
            if(!MakePlan(agentGoal, &m_plan)) {
                //We completely failed to make a plan for this goal
                FailGoal(agentGoal);
            } else {
                //We've got a new plan. Update what goal we're pursuing
                SetLastGoal(agentGoal);
                newGoal = true;
                hasPlan = true;
            }
        } else {
            //If we're here, our previous plan should still be valid.
            hasPlan = true;
        }
    } while (!hasPlan);
    SetShouldReplan(false);
}
```

Figure 5: An agent's logic to determine the most important goal for planning purposes.

in agent planning in a game situation. As written, the system handles creating and executing plans for whichever goal is most important to an agent, reevaluating for failed plans or plans that are no longer relevant to the current situation, and the creation of agents with varying actions and goals. The system is similar to that described in [21] and others, and is a suitable place to start making improvements upon the algorithm.

12 Improving the planner

Several improvements were made to the existing planning system. They vary in technique and required effort, but all allow greater flexibility for designers or improved behaviors in agents.

12.1 Scheduling

One of the unique aspects of partial-order planning systems is how (or in many cases, if) they create a totally-ordered plan from a complete partially-ordered plan. Since a single partial-order plan has many totally-ordered instantiations, the resultant totally-ordered plan could vary widely between planners. Several partially-ordered plans use *scheduling* to determine the order of actions or groups of actions in the ultimate total-order plan. Scheduling is usually denoted as a precedence ordering between actions, as in "Action A should occur before Action B." While this language immediately brings to mind the ordering enforced by a planner for establishing preconditions between actions, scheduling here means a less strict way of ensuring action ordering. For example, a

scheduling rule could be used to make sure that a "ReloadWeapon" action occurs before a "FireWeapon" action. Without scheduling, the only way to make such an ordering occur would be to put a precondition on the "FireWeapon" action that is established as a result of "ReloadWeapon." However, this could lead to extra processing as the "ReloadWeapon" must always be run as part of the plan to attack an opponent. With scheduling, the correct ordering will occur whenever the two actions are part of the same plan without forcing the "ReloadWeapon" action to be in the plan unnecessarily.

Scheduling is a rather simple addition to the planning system. Each action specifies a list of precedence relations, indicating the actions that it should occur before or after, as needed. For the purposes of demonstrating the feature, this project only implemented precedence for actions occurring before some other action. Once these relations are identified, ordering can be done after a plan is finished or during the creation of the plan itself. Since A* produced totally ordered plans to begin with, putting actions into the correct order at plan creation could avoid more expensive searches performed after all actions are in the plan. Care must be taken that reordering the actions does not undo any orderings required by the plan itself. This can be ignored, though, if the precedence relations put into place by an action avoid indicating any actions whose preconditions or effects have the same category as preconditions or effects on the original action. For example, an action whose effects include a change to the `kHasItem` WorldProperty should avoid stipulating a scheduling precedence relation with any action whose preconditions or effects include a `kHasItem` value. Figure 6 shows the algorithm used to insert actions into the correct precedence relation order during plan creation.

12.2 Goal merging

One of the central ideas put forth in [25] was combining the effects of several plans over several goals into lists denoting the particular effects of each possible plan, then merging the results. While this idea holds great promise for increasing the efficiency and believability of behaviors in artificial agents, one of its major restrictions is its need for several goals, or at minimum, several plans for a single goal. Certainly some games could benefit from the inclusion of such an algorithm, but for the games currently using planning algorithms, the requirement of several plans places too high of a computational price to make using the algorithm worthwhile. A similar result at a lower cost can be obtained through goal merging.

Goal merging is an algorithm developed to take two compatible goals and produce a single plan accomplishing both of them. As discussed in the research summary earlier,

```
if(! action->GetActionsToPrecede().empty())
{
    //If there are actions this one is supposed to precede, check to see if it's already in the plan
    bool didInsertAction = false;

    ActionContainer::const_iterator i = action->GetActionsToPrecede().begin();
    for( ; i != action->GetActionsToPrecede().end(); ++i)
    {
        ActionContainer::iterator actionToCheck = std::find(m_actions.begin(), m_actions.end(), *i);
        if(actionToCheck != m_actions.end())
        {
            //we found an action in the list that this one was supposed to be in front of
            ++actionToCheck;
            m_actions.insert(actionToCheck, const_cast<Action *>(action)); //put our new action right after the action we found
                //remember that these plans are in reverse order
            didInsertAction = true;
            break;
        } else {
            m_actions.push_front(const_cast<Action *>(action));
            didInsertAction = true;
            break;
        }
    }

    if(!didInsertAction)
    {
        m_actions.push_front(const_cast<Action *>(action));
    }

} else {
    //Otherwise, just stick this action at the front of the list.
    m_actions.push_front(const_cast<Action *>(action));
}
```

Figure 6: Algorithm to place actions into the correct order based on scheduling precedence relations.

goals can interact in a positive or negative fashion, or they could not interact at all. The algorithm merely takes two goals that positively interact or fail to impact each other, and returns a goal with mutually compatible target world states, such that a plan accomplishing the merged goal will accomplish each of the original goals.

The savings in the resultant plan depend entirely on the state of the goals that are merged. If the goals do not interact, the resultant plan cannot be any shorter than two independently generated plans for each separate goal. In the case of positively interacting goals, the resultant plan would be shorter than executing plans for each goal separately. This does not seem to have an immediate impact, though, as whatever mutually desired result will be accomplished by the execution of the first plan, shortening the plan for the second goal! However, in practice, actions can have multiple effects, and it is not always the case that a result accomplished during the execution of the first plan will still hold by the time the second plan is executed. Further, there is no wasted time processing a new plan after the completion of the first plan.

Negatively interacting goals can be detected by having mutually incompatible target states. For the purposes of this system, incompatible target states were defined as any target state with the same key type (`kHasItem`, for example), but a different value. Note that this could be overly restrictive in the case of floating-point valued target states, but it was an acceptable tradeoff for this system. It was further useful to allow a designer to specify certain goals as never being compatible; for example, a

```
//Takes two goals and returns one goal with the targetstates of both, if the goals are compatible.
// If the goals have incompatible target states (defined as at least two target states with the same key, but different value)
//  then MergeGoals returns firstGoal, and merged will == false.
// If the goals were merged, merged == true
Goal Goal::MergeGoals(const Goal *firstGoal, const Goal *secondGoal, bool& merged)
{
    GoalNameContainer::const_iterator i;
    i = std::find(firstGoal->m_exclusiveGoals.begin(), firstGoal->m_exclusiveGoals.end(), secondGoal->GetName());

    if( i != firstGoal->m_exclusiveGoals.end() )
    {
        //These goals are exclusive to each other
        merged = false;
        return *firstGoal;
    }

    Goal mergedGoal(*secondGoal);

    //  This double loop could be sped up considerably by keeping a hashtable of target states and values as we go through each list
    // However, so far the goal lists have been so short ( < 3), it isn't currently worth the overhead, and we're not
    // even coming close to going over our allotted frame time
    PropertyList::const_iterator first, second;
    for(first = firstGoal->m_targetStates.begin(); first != firstGoal->m_targetStates.end(); ++first)
    {
        bool skip = false;
        for(second = secondGoal->m_targetStates.begin(); second != secondGoal->m_targetStates.end(); ++second)
        {
            if(*first == *second)
            {
                //the goals have a matching target state, so skip processing this step
                skip = true;
                break;
            }
            // check for incompatible ending states
            //If these goals have a target state that is of the same type but different value, they're incompatible
            else if(first->m_key == second->m_key) //we know *first != *second because of the first if statement
            {
                merged = false;
                return *firstGoal;
            }
        }

        if(skip) {
            continue;
        }
        //At this point, we've got a target state (first) that's not in our second goal, so add it
        mergedGoal.AddEffect(*first);
    }

    merged = true;
    return mergedGoal;
}
```

Figure 7: Algorithm to merge two goals into a single goal.

"Retreat" goal might never be allowed to combine with an "Attack" goal. Having an early-out check in the algorithm could potentially save a lot of time in processing goals that would have ended up being incompatible in any case.

Figure 7 shows a listing of the algorithm used to merge two goals into a single target goal. In the case of incompatibility, a flag is set indicating that the goals failed to merge. As noted in the code comments, it would be possible to improve on the speed of this algorithm by keeping the goal's target states and values in keyed hashmaps, rather than searching lists for compatible or incompatible items. However, in the tests run for the system, most goals had so few target states that this was deemed an unnecessary optimization.

One drawback of this approach is the large amount of time searching on goals that are non-interacting. This would be reduced but not eliminated by implementing the hashtable optimization above. Another shortcoming of this method is that no plan is generated until after goal merging is complete. This means that an agent wishing to use goal merging would need to wait until all processing is complete on goal merging

before even starting to plan for attainment of the goal. With small goals and plans, this may not be an issue, but careful consideration should be taken before deciding to use this method.

12.3 Plan merging

Plan merging refers to the process of taking several independently generated plans and creating a single plan out of them, usually with the intention of reducing the overall cost of the plan. Often, a reduced-cost plan has the benefit of also producing more rational-looking behavior. To demonstrate the power of plan merging, an example will be examined before getting into the details of the algorithm.

Suppose an agent has the task of collecting items from around the world and returning those items to a home base. If the agent can only carry one item at a time, then it is apparent that it has no better choice than to go to an item, collect it, and return to base. However, if the agent can carry multiple items, it is also evident that many situations exist where the agent could reduce its total distance traveled by collecting several items at once. There are several ways we could accomplish this behavior utilizing a planning system. Suppose that the goal of collecting items and returning them to base was the "ReturnItems" goal. One could write a "GatherItems" action that accomplishes that goal. An agent executing the "GatherItems" action would look for the nearest items, gather as many as it could, and return them to base. While this is an acceptable solution, it is clear that our "GatherItems" action would be quite complicated. It would need to include code to pathfind and travel between items, pick up items, pathfind and travel back to base, and drop off the items once arrived. The increased functionality contained within one action works to defeat the purpose of having a flexible planning system. It is much easier to write smaller, reusable, atomic actions, such as GoTo for pathfinding, GetItem to gather the item from the world, and ReturnItem to drop off the item at base. These multiple actions allow the planner to do the complicated work of stringing together the actions into the right order, and further allow reuse of actions among many types of NPC's. Yet none of these actions can communicate to the agent that it should try to gather multiple items at a time. Instead, the agent can accomplish the desired behavior through plan merging.

The general idea is to take two plans with some overlapping actions, and combine the plans to produce a single plan with a lower cost than independently executing each of the original plans. In the given example, the agent could plan to gather each item independently, producing two plans that were unrelated but very similar, each with its own separate instance of a "GetItem" action. A possible result from a merge of those

two plans would combine as many possible actions together, producing a single plan with fewer actions. When the agent executes this plan, it collects both items before making the return trip to base.

12.3.1 Implementing a plan merging algorithm

Academically, the interest in plan merging centers mostly on plan optimization. [8] points out two major components to optimizing a plan: finding actions that can be merged, and then computing the optimal way to merge the actions if there exists more than one way to put the operators together. It is easiest to deal with these problems separately, so that is the approach taken here.

The first challenge in finding mergeable actions is discovering precisely what kinds of actions can be merged. Put simply, any number of actions can be merged together if there is another action that can replace the merged actions with 1) the same useful effects, and 2) the replaced action costs less than the sum of the merged actions it is replacing. Effects are defined as "useful" if they directly establish a precondition of another action in the plan, or a precondition of the goal itself. For example, suppose an agent has a plan to destroy a target and has the FireWeapon and ReloadWeapon actions at its disposal. The ReloadWeapon action has several effects: first, it makes the weapon be loaded, and second, it reduces the agent's ammunition store. The first effect is a useful effect, as it accomplishes a precondition of another action in the plan. The second effect isn't useful, as it has no ultimate bearing on the execution of the plan. While the effect may matter to the agent and may need to be considered in the creation of the plan, it can safely be ignored for plan merging.

Searching plans for mergeable actions would be incredibly expensive without knowledge of the actions themselves, so it is most efficient to specifically look for actions that are known to be mergeable. In an implemented system, this means either looking for a specific action that can be merged with itself, or looking for a known combination of actions that could be merged. In the earlier resource-gathering example, it is known that our agent is likely to have multiple plans, each with an instance of the "ReturnItems" action. This is an excellent candidate action to look for, since merging two ReturnItems actions together is known to be possible. In this specific case, the algorithm might even start its search at the end of the plan, since the ReturnItems action is likely to be the last action in each of the plans that are being merged. GoTo(Base) can similarly be merged with itself, as it obviously accomplishes the same effect.

The second challenge is creating an optimal plan once a possible merge has been discovered. [8] deals with the difficulties of creating an optimal plan, noting that creat-

ing an optimal plan quickly becomes computationally expensive, and is thus probably overkill to pursue in games. For the resource gathering NPC, its behavior is already much improved simply by allowing the agent to collect multiple resources at once. Rather than spend time worrying about the optimality of the plan, an algorithm could just place the rest of the two plans together. If more intelligent-looking behavior is desired or required of the system beyond a simple plan merge, an algorithm could employ *critics*, special-purpose checks used to help order any actions remaining after the merge. In our resource gathering example, it is known that there are two pairs of GoTo(Item) and Get Item actions that need to be placed before the merged ReturnItems action, so a critic could be written and employed to make sure the agent goes to the closest item first. Critics are general rules written to enforce a desired behavior in plan merges. The specific design and implementation of critics depend highly on the system for which they are needed.

At its simplest, then, a plan merging algorithm accepts two plans generated through the general-purpose A* planning system. An agent sends its two most important goals to the planner, for example, and then sends those two independently generated plans to the plan merger. For every action in the first plan, the algorithm checks to see if it can be merged with an action in the second. If a merge can be performed, those two actions are put together into a single plan, being careful to put preceding actions from both plans before the merged action, and likewise putting any actions occurring after the merged action afterwards. If more precise control over the order of the non-merged actions is needed, critics can be employed to determine the best ordering and rearrange the actions as necessary. For a wider range of possible merges, a complete plan merging algorithm should examine the net effects of every possible group of actions in each plan, looking for situations where a sequence of actions could be replaced by a single cheaper action. Such an algorithm produces the most impressive improvements to mergeable plans, but is also expensive to run. An example of an algorithm checking for merges on a particular known action is shown in Figure 8.

12.3.2 Beyond single-agent merges

While merging two plans for a single agent certainly offers opportunities for improved behavior, plan merging also offers remarkable benefits in the areas of squad-based planning. For instance, an agent utilizing plan merging could merge an individual goal (picking up a weapon or health power-up) with a squad-issued goal (providing cover fire). Utilizing plan merging in these situations allows an agent to maintain its own goals and personality in the face of squad-issued orders and even allows for situations

```
Plan Plan::MergePlans(const AI::Plan &firstPlan, const AI::Plan &secondPlan, bool& merged)
{
    merged = false;

    //Here, we take the strategy to look for actions we know we can merge
    ActionContainer::const_iterator firstPlanIter, secondPlanIter;
    for(firstPlanIter = firstPlan.m_actions.begin(); firstPlanIter != firstPlan.m_actions.end(); ++firstPlanIter) {
        if((*firstPlanIter)->GetName() == "ReturnItemAction") {
            break;
        }
    }
    if(firstPlanIter == firstPlan.m_actions.end()) {
        return firstPlan;
    }

    for(secondPlanIter = secondPlan.m_actions.begin(); secondPlanIter != secondPlan.m_actions.end(); ++secondPlanIter) {
        if((*secondPlanIter)->GetName() == "ReturnItemAction") {
            break;
        }
    }
    if(secondPlanIter == secondPlan.m_actions.end()) {
        return firstPlan;
    }

    //If we make it here, we can merge the plans on ReturnItemAction
    Plan mergedPlan;
    mergedPlan.m_actions.insert(mergedPlan.m_actions.end(), firstPlan.m_actions.begin(), firstPlanIter);
    mergedPlan.m_actions.insert(mergedPlan.m_actions.end(), secondPlan.m_actions.begin(), secondPlanIter);
    mergedPlan.m_actions.push_back(*firstPlanIter);
    mergedPlan.m_actions.insert(mergedPlan.m_actions.end(), ++firstPlanIter, firstPlan.m_actions.end());
    mergedPlan.m_actions.insert(mergedPlan.m_actions.end(), ++secondPlanIter, secondPlan.m_actions.end());

    merged = true;
    return mergedPlan;
}
```

Figure 8: A plan merging algorithm with a known action to merge on.

where the agent can accomplish many goals at once.

A plan merging algorithm for such an application has no discernible differences from the one described for an agent pursuing two self-generated goals. However, as the variety of possible goals and actions increase, a generalized version of the algorithm that checks for compatible combinations of useful effects between actions becomes more important for the overall success of the merging operation.

12.3.3 Strategies for improving action searching

Searching two or more plans for actions with similar effects is expensive, especially if the merger considers replacing groups of actions with different net effects. If the game that the merging operator is developed for is fast-paced, typical of many FPS's, an agent's primary and secondary goals could change more quickly than it could even devise a plan for its secondary goal. Clearly, plan merging is of no use unless we can quickly perform the merge.

One possible strategy to reduce the time needed to search through actions is to only look for mergeable actions when specific actions are present in the plan, something that can be determined in the middle of the plan-making process. For extremely long plans, hooks direct to possibly-mergeable actions could be included in the plan structure itself, directing the algorithm not only into the correct places immediately, but also informing

it if a merge is worth looking for at all. In specific kinds of agents, it might even be worth only looking for a specific action to merge in each plan. In cases where the agent's primary goal does not contain any mergeable actions, the planning process for the secondary goal can be entirely avoided.

Similarly, we might only attempt a merge when the goals being planned for are compatible. Conversely, it makes sense to not even bother to attempt a merge if the two intended goals are incompatible. Indeed, even making a plan for a secondary goal is wasted time if our goals are incompatible. This determination is probably best made by the programmer, much as was described above in the section on goal merging. It may be obvious to us that an Attack and a Retreat goal will never produce mergeable plans, but a generically written algorithm would search through every action of each plan before reporting that no mergeable actions exist. This extra work can be avoided with a little extra initial effort on the part of designers and programmers.

13 Conclusions and future work

Plan merging is a clear method of improving the perceived intelligence of agents in interactive simulations. Based on the variety of methods available to perform merging, a suitable plan merging algorithm balancing a variety of mergeable actions with runtime complexity can be chosen for nearly any situation imaginable. Plan merging is a more flexible merging alternative to goal merging, although the latter can be performed more cheaply in the general case. In either situation, planners can be improved at not much cost in order to improve planning results.

Different plan merging algorithms may be better suited to different situations. The algorithms described in [25] and [26] are not well suited to fast-acting agents, but may prove useful for agents acting over longer periods of time, such as in strategy games. These situations remain to be formally examined.

Several improvements are readily suggested for the system as described. It would be beneficial to determine a more general solution to determine when goals are incompatible with each other, especially with similar goal effects on real-valued world properties. Further, all of the actions and goals are defined within C++ classes. While this was acceptable for the purposes of this research, in a professional game development environment there would be a greater emphasis on developing a reusable script or tool for both designers and programmers to more readily tweak and create actions.

References

[1] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[2] Craig Boutilier and Ronen I. Brafman. Partial-order planning with concurrent interaction actions. *Journal of Artificial Intelligence Research*, 14:105–136, 2001.

[3] The Duy Bui and Wojciech Jamroga. Multi-agent planning with planning graph. In *Proceedings of eunite*, 2004.

[4] P. R. Cohen, C. R. Perrault, and J. F. Allen. Beyond question answering. In W. G. Lehnert and M. H. Ringle, editors, *Strategies for Natural Language Processing*, pages 245–274. Erlbaum, Hillsdale, NJ, 1982.

[5] Ken Currie and Austin Tate. O-Plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[6] Kutluhan Erol, James A. Hendler, and Dana S. Nau. Semantics for HTN planning. Technical Report CS-TR-3239, Institute for Systems Research and Institute for Advanced Computer Studies, 1994.

[7] Kutluhan Erol, James A. Hendler, and Dana S. Nau. UMCP: A sound and complete procedure for hierarchical task-network planning. In *Artificial Intelligence Planning Systems*, pages 249–254, 1994.

[8] David E. Foulser, Ming Li, and Qiang Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57(2–3):143–181, 1992.

[9] C. Geib and R. Goldman. Plan recognition in intrusion detection systems, 2001.

[10] Daniel Higgens. Generic a* pathfinding. In Steve Rabin, editor, *AI Game Programming Wisdom*. Charles River Media, 2002.

[11] Daniel Higgens. How to achieve lightning fast a*. In Steve Rabin, editor, *AI Game Programming Wisdom*. Charles River Media, 2002.

[12] D. Isla. Handling complexity in the Halo 2 AI. In *Game Developer's Conference Proceedings*, 2005.

[13] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, August 1992.

[14] Craig A. Knoblock. Generating parallel execution plans with a partial-order planner. In *Artificial Intelligence Planning Systems*, pages 98–103, 1994.

[15] Wenji Mao and Jonathan Gratch. Decision-theoretic approach to plan recognition. Technical report, Institute for Creative Technologies, 2004.

[16] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*, pages 373–400. Morgan Kaufmann Publishers, Inc., San Francisco, 1998.

[17] Jeff Orkin. Applying goal-oriented action planning to games. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 217–227. Charles River Media, 2004.

[18] Jeff Orkin. Symbolic representation of game world state: Toward real-time planning in games. *AAAI Challenges in Game AI Workshop Technical Report*, 2004.

[19] Jeff Orkin. Agent architecture considerations for real-time planning in games. In *AIIDE Proceedings*, 2005.

[20] Jeff Orkin, 2006. Personal communications.

[21] Jeff Orkin. 3 states & a plan: The A.I. of F.E.A.R. In *Game Developer's Conference Proceedings*, 2006.

[22] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 103–114. Morgan Kaufmann, San Mateo, California, 1992.

[23] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, first edition, 1995.

[24] Austin (Ed.) Tate and James (Ed.) Hendler. *Readings in Planning*, pages 291–296. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[25] John Thangarajah, Lin Padgham, and Michael Winikoff. Detecting & exploiting positive goal interaction in intelligent agents. In *AAMAS '03: Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, pages 401–408, New York, NY, USA, 2003. ACM Press.

[26] John Thangarajah, Michael Winikoff, Lin Padgham, and Klaus Fischer. Avoiding resource conflicts in intelligent agents. In *Proceedings of the 15th European Conference on Artificial Intelligence*, 2002.

[27] Neil Wallace. Hierarchical planning in dynamic worlds. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, pages 229–236. Charles River Media, 2004.