DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDY PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS OF Brian Cooley-Gilliom

HAS BEEN SUCCESSFULLY COMPLETED ON 6/6/2012

TITLE OF THESIS: **INCREASING THE VARIETY OF AGENT BEHAVIOR USING PROBABILISTIC PLANNING PLANNING**

MAJOR FILED OF STUDY: COMPUTER SCIENCE.

COMMITTEE:

[illegible]
Dmitri Volper, Chair

[illegible]
Gary Herron

[illegible]
Xin Li

[illegible]
Benjamin Ellinger

APPROVED:

[illegible] 6/6/12
Dmitri Volper date
Graduate Program Director

[illegible] 6/15/2012
Xin Li date
Dean of Faculty

[illegible] June 15, 2012
Samir Abou-Samra date
Department Chair of Computer Science

[illegible] 6/18/2012
Claude Comair date
President

INSTITUTE OF DIGIPEN INSTITUTE OF TECHNOLOGY

PROGRAM OF MASTER'S DEGREE

THESIS APPROVAL

DATE: 6/6/2012

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE THESIS PREPARED BY

Brian Cooley-Gilliom

ENTITLED

Increasing the Variety of Agent Behavior Using Probabilistic Planning

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE FROM THE PROGRAM OF MASTER'S DEGREE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

[illegible]

Dr. Volper
Thesis Advisory Committee Chair

[illegible]

Dr. Volper
Director of Graduate Study Program

[illegible]

Dr. Xin Li
Dean of Faculty

Increasing the Variety of Agent Behavior Using Probabilistic Planning

BY

Brian Cooley-Gilliom

B.A. Computer Science, B.A. Film and Digital Media, UC Santa Cruz

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

Spring 2012

Table of Contents

List of Figures

1 Introduction

As games have become increasingly large and difficult to manage, many game studios have moved away from Finite State Machines (FSM) in favor of techniques that reduce complexity for game designers and can more effectively handle dynamic environments. When using FSMs, game designers must constantly restructure the states and transitions to accommodate new behavior. Additionally, behaviors produced by FSMs are typically reactive rather than proactive. Ideally, agents should be able to solve problems based on their goals rather than make one short-term reaction to the immediate state of the world. One of the most interesting areas of study which attempts to solve this problem is referred to as “planning”.

Current planning approaches in video games allow an agent to dynamically solve problems in a real-time environment. However, these approaches assume agents have complete information about the world and that their actions will always affect the environment in the expected way. Additionally, a plan is either a solution or not; once a solution is found, there is no consideration given to how effective it may (or may not) be. In order to change the way an agent approaches a problem, a game designer must add or remove actions available to the agent. In the real world, many factors influence the quality of one’s solution to a problem. One might be taken by surprise, be distracted by multitasking, or simply not be skilled in solving problems. In video games, interesting agents should appear intelligent, but still make occasional mistakes that allow the player to prevail with enough skill. Agents with the same set of skills and circumstances, but a different desired difficulty, should behave differently. An enemy in a video game should use more efficient and careful strategies instead of simply moving more quickly and being more resilient.

This paper presents an application of the Buridan probabilistic planner (Weld, Hanks, & Kushmerick, 1994) to a video game environment. The objective is to take probability of success into account, while still favoring efficient plans and retaining game designer convenience, and to show that probabilistic planning can be used to create interesting agents which are able to solve problems in a variety of ways such that their behavior realistically affects the difficulty for the player.

2 Background

In general terms, planning algorithms empower agents to solve problems using their abilities. The difference between planning and most other commonly used techniques in video game artificial intelligence, such as FSMs and behavior trees, is the way an agent's goals and abilities are used in relation to one another. In FSMs, an agent might have one or more FSMs describing its behavior. Each state represents what the agent is currently doing. An agent in a game, for example, could be in either the **Patrolling** or **Attacking**. Transitions, then, represent what the agent should do next, given a new piece of information. If, while the agent is **Patrolling**, an enemy passes by, the FSM would likely transition to the **Attacking** state. For many applications, FSMs are perfectly suitable and are very simple to implement. In complex problems, however, they become brittle and increasingly likely to become a source of bugs and unexpected behavior (Orkin, 2006).

In the video game development process, the artificial intelligence (AI) design is often a fluid process. New behavior must often be added that was not originally planned (Orkin, 2006). Modern games have complex dynamic environments, in which the player can quickly change the state of the game world. Agents must be able to react to many types of situations with appropriate actions, resulting in many possible states with many possible transitions to new states. Adding a new ability to an agent means adding at least one new state. If this ability can be used at any time, the agent must be able to transition to this state as well as transition away to some number of other states. Consequently, in the worst case, an agent with *n* possible states, has *n(n-1)/2* possible transitions (i.e. a complete graph). Adding a new state could potentially mean adding *2n+1* new transitions, each with its own conditions. Similarly, if a new aspect of the game environment must be taken into account, any relevant state must either be given a new transition or one of its transitions must be changed to take the new information into account. This amount of work makes FSMs ripe for mistakes because of the spaghetti-like logic.

Giving an FSM the ability to solve a new problem is, therefore, unavoidably interlocked with anticipating each possible circumstance and attempting to pre-build the functionality to handle all of them. Planning algorithms attempt to remove this rigid coupling between actions, goals, and the state of the world. Generally, the input to a planner is the current state of the world, the actions available to the agent, and a desired goal state. A planner then returns a series of actions which the agent can take to achieve its goals.

2.1 Planning Terms and Definitions

There are several important concepts that are common to the planners discussed in this paper. Plans are essentially a transformation from one state to another, and generally speaking, all planners use the same basic building blocks to construct their plans. The most basic parts are objects and literals. Objects are the elements of the world that the planner can interact with. Literals are simple statements that evaluate to either true or false and often use objects in their definition. For example, if the world description has the objects *agent* and *crate*, it may also have the literal ***At***(*agent, crate*). This literal evaluates to true if the agent is positioned at the crate, and false if not.

The term 'world state' will often be used in the discussions to come. A state describes the situation of the world at a certain time. Here, a state will be defined as a set of literals, which describe relationships between various objects and entities in the world. The time step in a plan, t, is defined in terms of the number of actions which have been executed so far. If two or more actions are executed concurrently, it only counts as a single time step. Each plan starts with an initial state and a goal state. A successful plan uses actions to incrementally transform the initial state into the goal state, and, consequently, time is not measured in seconds but rather as a position in the plan, where the initial action is occurs at $t = 0$. Subsequent actions occur at t = 1…N, where the goal state always occurs at t = N. The world state at t = n is found by starting with the initial state and transforming it by executing the actions that occur at t = (0, n).

Actions represent the abilities of an agent. While the structure of actions will vary depending on the planner, for now it is sufficient to generalize. Each action is made up of a set preconditions and a set of effects. The set of preconditions define the world state necessary for the action to be executed. The set of effects defines the changes that will be made to the world state when the action is executed. A planning solution consists of a series of actions that transforms the initial state such that the preconditions of the goal state are met.

At the heart of each planning algorithm is a search of some kind. Most of the algorithms mentioned here use some kind of guided search. Specifically, the algorithms discussed in depth utilize an A* search, so it is useful to briefly review the high level functionality of this algorithm. The A* search algorithm, unlike a breadth-first search (Russel, 2003), attempts to consider more promising nodes first, so that each node is ranked with a heuristic and a cost. Only if a potentially cheap solution is found to be invalid does the planner back out and consider more costly solutions (Russel, 2003). The planners discussed here use A* to search plan-space. Each node in the search tree is a representation of an incomplete plan. When an incomplete plan is examined, each change, or refinement, is thought of as a child plan. Different planners use different representations and heuristics, but each is evaluated in an analogous way. The cost at a particular point represents how efficient a plan is, while the heuristic is an estimate of how close an incomplete plan is to a solution.

The actual search process is as follows: Starting with the initial state, new child plans are created and added to the open list. The open list contains nodes which have not yet been examined, and is sorted based on the cost and heuristic score of the nodes. At each step, the topmost node is removed and examined. Each change made to the current plan results in a new child plan, each of which is scored and added to the open list. This process continues until a solution is found.

2.2 Types of planners

While the vast majority of planners share the common general goals and structures described so far, individual planners differ both in the types of problems they attempt to solve and the specific methods they use to solve them. In order to choose the correct starting place for the goal of planning for variety, it is important to examine the strengths and weaknesses of each family of planners. Additionally, since finding an interesting variety of plans is somewhat contrary to finding the best possible plan, each approach must be examined with respect to the specific goals in a planning problem. The fastest, state-of-the-art planner is not necessarily the best solution for the goals of this project.

2.2.1 Deterministic Planning

Deterministic planners optimistically assume that each action in a plan will always have the desired effects and that there are no unanticipated outside events that could potentially invalidate a plan. So far, all disclosed applications of planners in video games have been deterministic, so it is necessary to examine their strengths and evaluate their potential for producing a variety of plans. The most important feature of deterministic planners is that each action is assumed to execute successfully without consideration for unanticipated outside influences that may occur during the execution of a solution plan. Within the class of deterministic planners, it is also important to distinguish between sequential and non-sequential planners.

2.2.1.1 Sequential Planning

Sequential planners are the simplest type of deterministic planning algorithms. As actions are added, it is assumed that actions do not interfere with one another. While this limits the problem solving capability, some are simple enough to not require the consideration of such problems. Accepting these limitations, sequential planning problems can be solved extremely quickly. Additionally, video game applications of sequential planning have many desirable properties that should remain present in the planner presented here.

The Stanford Research Institute Problem Solver (STRIPS) (Nilson, 1998) was the first planner to gain traction in the AI community. By performing a backwards search, plans are built by adding actions that allow an agent to reach its goal. Each new action added as the plan is built is assumed to always occur before actions that are already in the plan. The world in the STRIPS planner is represented by Boolean literals. The goal of the planner is to transform the world from an initial state to a goal state by adding actions. The goal state is defined as a set of literals that must evaluate to true for the plan to be considered valid. At the beginning of the process, these literals are added to a list of open preconditions. As the planning process progresses, literals will be added and removed from this list. Each action in the STRIPS planner has an “add” and “delete” list (here these are referred to as preconditions and effects). They are so named because when an action is added the plan, its effects are deleted from the list of open preconditions, and its preconditions are added to the list. If, at any point, a precondition is satisfied by the initial world state, a new action need not be added. This process continues until the set of open preconditions is empty.

The limitations of STRIPS, resulting from its deterministic basis, make it inadequate for simulating most real-world situations. However, video games, which have the obvious advantage of not requiring the simulation of the real world, have successfully used the STRIPS planner (with some modifications) to create a useful and powerful tool for agent design and dynamic problem solving. F.E.A.R. (Monolith 2007) employs the STRIPS planner in its AI system, named Goal Oriented Action Planning (GOAP). GOAP has several objectives: flexibility, dynamic problem solving, realism, and ease of use. The cornerstone of the project is the difference between reactive and goal-directed behavior. When using FSMs, one must consider what the agent will do for each possible circumstance from each possible situation. In goal-directed behavior, the AI module is sent the current state, the goal state, and a set of available actions to take. This decoupling enables more efficient and flexible designs and better represents the way humans approach problems. Changing the behavior of agents can be accomplished by adding or removing the actions available to the agent. Since the actions are not necessarily specific to a particular agent, these actions can potentially be reused across multiple agents and games. In F.E.A.R., this system was used to control high level behavior, such as an agent figuring out how to get into a locked room, and low level behavior, such as which animation to play at a given time.

In GOAP, agents can select one of several actions to accomplish the same goal. This requires that the planner has a logical way to choose between multiple actions. In order to solve this, GOAP allows the designer to assign a cost to each action. Much like path finding with A*, cheaper plans are looked at first for further investigation and refinement. If the planner is unable to refine the cheaper plan into a valid solution, the more expensive plans eventually rise to the top of the open list. In this way, agents take the most efficient course of action available to them, much like A* path finding takes an optimal path to a goal location.

In F.E.A.R., actions and goals are assigned to various agent types using an editor. Each agent has a unique set of actions and goals, which are listed in order of priority. If a plan can be found that satisfies the topmost goal, the agent will take those actions. Otherwise, it will carry out lower priority goals in order of priority. For example, an agent in a room may have the goals **AttackPlayer** and **PatrolRoom**. Until the player is present, the agent will carry out the plan found to satisfy the **PartrolRoom** goal. Once the world state is such that a valid plan can be found for the goal **KillPlayer**, the agent will take the actions defined in the newly found plan. By assigning agents different actions which can accomplish the **KillPlayer** goal, a designer can create unique behaviors. The agents **Soldier** and **Assassin**, for instance, are given different abilities. The soldier walks around the room until it detects the player. The agent is then able to find a valid plan for the **KillPlayer** goal by using the **FireWeapon** action. The assassin, however, hides until the player is near, and then uses the **MeleeAttack** action to satisfy the **KillPlayer** goal.

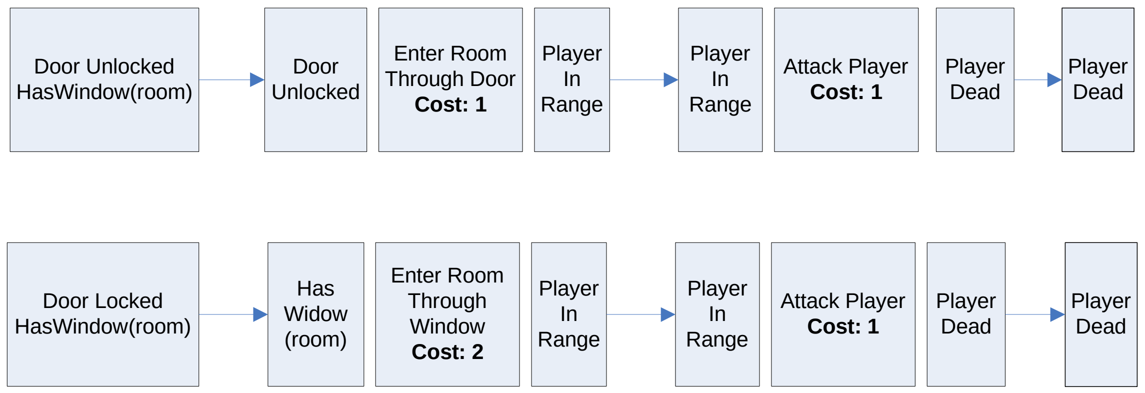

Figure 1: Example of a plan generated by G.O.A.P.

The agent attempts to enter through the door first, as it is cheaper (costs 2). If the door is found to be locked, the agent falls back on the more expensive plan (costs 3).

Another important impact of GOAP on AI behavior is re-planning. In another example, illustrated in Figure 1, an agent must enter a room to get to the player. In the first attempt, the agent walks to the door, opens it, and attacks the player. In the same example, with the player holding the door (Door Locked), the agent's first attempt fails. The next attempt involves kicking down the door which also fails. In the third attempt, the agent uses the **EnterRoomThroughWindow** action and is able to satisfy the **KillPlayer** goal. In this example, re-planning highlights the difference between goal-directed and reactive behavior, and in this case, allows the player to observe a human-like trial-and-error problem solving behavior.

Sequential planning has also been applied to real-time strategy (RTS) games. Chan et. all (2007) present an algorithm to make resource production decisions. Their algorithm starts with the STRIPS planner, which is used to determine which actions need to be taken. Often in RTS games, there are dependency trees that dictate requirements to build certain items. To build a soldier unit, for example, the player must build a barracks. To build a barracks, the player must have a builder unit and the required resources. Once the planner determines which actions are necessary to accomplish the desired goal, the plan is sent to a scheduler. There, the plan is condensed to allow for concurrent actions so that actions that are not interdependent can be executed at the same time. For instance, it may be possible to build an additional worker unit while another unit gathers resources and another building is being constructed.

The work of (Chan, Fern, Ray, Wilson, & Ventura, 2007), and scheduling algorithms in general, are outside the scope of this project, but provide inspiration for further use of planning in video game AI. While the game designers of F.E.A.R. are able to arrange actions and goals in such a way that different agents behave in unique ways, this variety must be explicitly specified and is thus limiting. Additionally, since likelihood of success is not considered by the planner, agents may take actions which lead to pitfalls from which they are unable to recover (a situation where re-planning is impossible will be referred to as a dead end). Finally, as discussed below, sequential planners inherently impose a limit on the problem solving capability of agents. For these reasons, the focus of this project will be more powerful planners.

2.2.1.2 Hierarchical Task Network Planning

Hierarchical Task Network (HTN) planning has also been used in recent games. HTN planning requires a designer-created network, in which high level actions are broken down into smaller, more specific actions (Kelly, Botea, & Koenig, 2007). A general goal can be solved in more than one way if the network properly describes it. For instance, a general PrepareDinner action could be broken into the sub-actions (GetPhone, CallForPizza, PayDeliveryGuy) or (GoToFridge, GetTVDinner, UseMicrowave). Actions can also recursively reference each other, allowing actions or sequences of actions to be repeated. During the planning process, the network is traversed and a series of actions which match the current world state is returned. Using a hand built network significantly reduces the size of search, but also increases the work load of the designer of the network. Like FSMs, a small change to agent behavior may require large changes to the design of the network, depending on the amount of interdependency.

HTN planning has been used both online and offline. The implementation presented by (Kelly, Botea, & Koenig, 2007) uses planning to build scripts, which can then be loaded into existing games that support scripting. This planner can be used to generate scripts for many non-playable characters (NPC). This allows designers to avoid tedious task of writing lengthy scripts, which, in this example, were used to control the daily actions (eating, sleeping, or working) of the NPCs in Oblivion (Bathesda, 2007). Killzone (Sony, 2009) uses online HTN planning to find plans for both individuals and squads in real time. When coupled with other modules of the AI, this planner yields successful results (Champandard, Straatman, & Verweij, On the AI Strategy for KILLZONE 2's Multiplayer Bots, 2010).

However, when compared to GOAP in terms of ease of use for the designer, HTN planning has some short comings. Building the network to solve non-trivial problems is similar to writing an algorithm. Thus, additions or changes to the problem and goals can require a substantial amount of work to build and maintain the HTN to ensure each case will be handled. Since the search is guided through the pre-built network, in order for the planner to produce a variety of different plans, the logic to do so must be added to the HTN by a designer.

2.2.1.3 Partial Order Planning

Partial-order planners (POP), often called Systematic Non-Linear Planners as first described by (McAllestor & Rosenblatt, 1991), were the first planners to successfully improve on sequential planners without relying on user-provided domain-specific information. Unlike sequential planning, actions added to a partial-order plan have as little constraint on when they may occur as possible. While an action is constrained to occur before an action it directly supports (i.e. an effect of the action is being used to support the precondition of another action), actions which are not interdependent cannot be assumed to be executed in any particular order. This method of refinement is often referred to as a least-commitment technique. By only making a definite choice when absolutely necessary, a POP is able to solve more complex problems than a sequential planner. The additional capability is achieved by constantly evaluating plans for conflicts during the planning process. A partial-order planner interleaves actions when necessary so that actions do not interfere with one another.

The Sussman Anomaly is a planning problem used to illustrate the limitations of a sequential planner and demonstrate the advantages of partial-order planners. "Block World" is a domain often used for testing planners that involves a number of blocks and a robotic gripper which must stack blocks in various ways. This example also includes the rule that, in order to move a block, it must not have another block on top of it. Three blocks, *a*, *b*, and *c,* are positioned such that *a* and *b* are on the table and *c* is on *a* (Figure 2). The goal is to stack *b* on *c* and *a* on *b*. To solve this problem, the robot must remove *c* from *a* and then stack the blocks in order. However, a sequential planner is not capable of solving this problem. The planner starts with the two preconditions **On**(*b,c*) and **On**(*a,b*). If the planner chooses to solve the precondition, **On**(*b,c*), it will stack *b* on *c* without first removing *c* from *a*. If the planner chooses the second precondition, **On**(*a,b*), the planner will move *c* to the table and then stack *a* on *b*. In either case, solving the next precondition would require undoing the previous step (Russell, 2003).

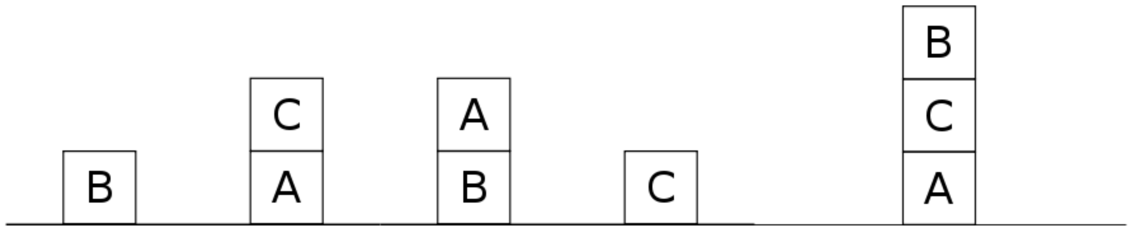

Figure 2: Sussman Anomaly problem states for sequential planners

From left to right: the initial state, solving the precondition On(b,c) first, solving the precondition On(a,b) first. In order to solve the next preconditions, On(a,b) or On(b,c), respectively, a sequential planner must undo the previous step, and will repeatedly oscillate between the middle and left states.

Partial-order planners can successfully solve problems like the Sussman Anomaly by interleaving the actions in plans. At a high level, a POP would solve this problem in the following way: First, the actions **Move**(*b, c*) and **Move**(*a, b*) are added to the plan to satisfy the goal condition. Note that, so far, these actions are not strictly ordered with respect to one another. Next, the planner adds the action Move(c, table) to satisfy the preconditions added by the action **Move**(*a, b*). As the **Move**(*c, table*) action directly supports the action **Move**(*a, b*), these actions are strictly ordered. However, the action **Move**(*b, c*) may still occur at any time (before **Move**(*c, table*), between **Move**(*c, table*) and **Move**(*a, b*), or after **Move**(*a, b*)). By identifying potential threats, the planner determines that the only valid place for the action is between the actions **Move**(*c, table*) and **Move**(*a, b*). The resulting solution is the sequence of actions **Move**(*c, table*), **Move**(*b, c*), and **Move**(*a, b*). Since the planner determines the ordering of actions rather than the order in which goal conditions are encountered, this problem is solvable by a POP regardless of input order.

Partial order planners also have limitations. Like the sequential planners discussed above, they do not take likelihood of success into account. Problems that require reasoning about how likely actions are to actually produce their desired effects are not solvable. This limitation is most easily illustrated with the bomb-toilet problem. In this example, a robot must prevent an explosion by dunking a bomb in a toilet. However, there are two briefcases, either of which could contain the bomb. To solve the problem correctly, the robot must dunk both bombs in the toilet. While the robot is certain to do extra work by this approach, the problem is solved with a 1.0 success probability. A partial-order planner is unable to recognize the possibility that the bomb could be in either briefcase, so the robot will only dunk one. The resulting plan only has a .5 success probability (assuming there is a 50/50 chance the bomb will be in either case). Thus, many partial-order planners have been expanded into more sophisticated versions to handle more difficult problems.

2.2.1.4 GraphPlan

The deterministic version of Graphplan (Blum & Furst, 1997) is the final deterministic planner evaluated in this study. Graphplan has the same shortcomings as other deterministic planners for these purposes, but is important to consider because there is a probabilistic version of the algorithm and planning graphs have become one of the most successful heuristics for evaluating partial plans during the search process.

Graphplan is a planning algorithm which uses the properties of a graph to make the planning process more efficient. The planner first constructs the graph, which can be done in polynomial time. The graph consists of alternating levels of propositions and actions. Each level represents a time step, and contains either the actions or propositions that are possible at that time. Each edge in the graph connects an action to a precondition or an effect. At this point in the process, no consideration is given to actions or propositions that could potentially interfere with one another. When constructing the graph, the algorithm starts with the initial conditions of the plan and constructs additional action levels until it is possible that all of the plan's goals are achieved. The result is a planning graph with t time steps. If a valid solution exists, it is guaranteed to be a sub-graph of the planning graph. If, after performing the actual plan search, a valid plan is not found, the plan graph is extended one time step. In this way, the planner either returns a valid plan or proves that no valid plan exists of the current length.

Finding a valid plan within the planning graph involves a backward-chain search. At each level, starting with the goal propositions, the planner looks for actions which can satisfy the preconditions. As the search progresses, actions and propositions which interfere with one another are recorded and the planner ensures that actions that are mutually exclusive cannot occur at the same time. If, at any level, the planner fails to find actions to support the open preconditions, the planner backtracks and attempts to find a different sequence of actions. If all sequences have been exhausted, the planning graph is extended by one time step in the aforementioned way.

2.2.2 Contingent Planning

Deterministic planners assume that the state of the world present at the time of planning will persist throughout the execution of the plan. Contingent, or conditional, planners as exemplified by Contingent-FF (Hoffman & Brafman, 2005), attempt to handle unexpected events into account by allowing the agent to observe the world during plan execution. Special actions that allow the agent to gather information are linked with two or more normal actions to create a branching structure. At the time of plan execution, the agent chooses a single action that will execute successfully in the current world state. While contingent planning solves some of the problems which arise in an uncertain environment, the same results can be achieved by monitoring plan execution and re-planning with a sequential planner (as in G.O.A.P.). Since sequential planning is of NP complexity and contingent planning is NP-Hard, it is generally better to generate additional sequential plans as needed rather than try to take all possible events into account at once. Additionally, in order to find an interesting variety of plans that directly influence the difficulty of an agent, we must have an idea of how successful a plan will be as a whole, which contingent planners are unable to provide. Thus, contingent planning, like sequential planning, does not have the necessary properties for purposes of planning variety and linking behavior to difficulty.

2.2.3 Conformant Planning

Conformant planning takes a different approach to account for uncertainty. Rather than observing the world as the plan executes, a conformant planner attempts to model uncertainty by describing multiple possible world states (Blythe, 1999). The planner then attempts to find a plan which works for all given world states. There are several varieties of conformant planning. The planner T0 (Palacios & Geffner, 2009) is limited to problems where uncertainty is only considered in the initial state of the world. Others, such as Conformant Graphplan (CGP) (Smith & Weld, 1998) and Conformant FF(CFF) (Brafman & Hoffmann, 2006), consider actions with multiple possible outcomes. Conformant planners can also differ in the possible solutions to planning problems. Pure conformant planning solutions require solutions which guarantee successful execution (Russel, 2003). There is only one solution which takes all known possibilities into account. Probabilistic conformant planners allow solutions to have varying degrees of success. Here, the former will be referred to as a conformant planner and the latter as a probabilistic planner. As conformant planning is generally significantly more efficient than probabilistic planning, it is important to examine the features of this family of planners and evaluate their usefulness to this project. While increased efficiency is always desirable, the solution to a problem will always be the same. Conformant planners attempt to model uncertainty about the world and execution of the plan in order to build more successful plans. However, since solution plans found by conformant planner are either solutions or not, they do not offer a way to fine tune results without changing the structure of actions. The planners that will be focused on for the purposes of this project are able to find solutions with varying probabilities of success.

2.2.4 Probabilistic Planning

Probabilistic planning is a specific type of conformant planning. Like conformant planning, features and problem definitions vary from planner to planner. Full probabilistic planners are able to solve problems where both initial world states and the effects of actions are defined as probability distributions, rather than single states or effects.

One of the first successful probabilistic planners was Buridan (Weld, Hanks, & Kushmerick, 1994), and is a direct extension of a partial order planner, Universal Conditional Partial Order Planner (UCPOP) (Penberthy & Weld, 1992). Buridan retains POP structure and adds support to reinforce preconditions in the plan with multiple causal links. Additionally, potential solutions are assessed and assigned a value of expected probability of success. In practice, the user provides a planning problem and a required threshold of success that a solution plan must meet. This planner has several features that are useful when trying to find a variety of plans. Different solutions can be found by changing a single number. Additionally, the heuristic function and the cost of actions are easily adjustable.

At first glance, two of the most promising options for probabilistic planning are PGraphplan and TGraphplan (Blum & Langford, 1999). PGraphplan and TGraphplan are two different approaches that extend CGP into a probabilistic planner. Like the deterministic version, both algorithms use a planning graph, but each uses a unique technique to find the actual plan. PGraphplan treats the planning problem as a Markovian Decision Process (MDP) and solves using a top-down dynamic programming approach and a forward search. PGraphplan produces optimal plans, but was found to be slower than TGraphplan in empirical tests. TGraphplan uses a search technique similar to that found in the standard Graphplan algorithm. Thus, it is able to take better advantage of the planning graph to shorten the search. However, the plans found by TGraphplan are not guaranteed to be optimal. In both cases, the planners attempt to find a plan with the maximum expected probability given a maximum time frame, which is defined as the number of action/proposition layers in the planning graph.

Both Graphplan-based planners have been shown to be substantially faster than classic POP based probabilistic planners (discussed below), especially in larger planning problems. Additionally, the problem solving capability is sufficient to solve problems in STRIPS style domains, which should allow for interesting problems to be solved for games. However, the required formulation of problems, such as giving a maximum time frame and producing a plan with maximum probability, does not allow for the fine control needed to generate a variety of plans from the same actions and circumstances. While adjusting the maximum time frame would give a designer some control over the resulting plans, it would be much more useful to specify the probability threshold directly. In most domains, such as robotics, it is unlikely one would ever require plans with a lower-than-possible probability of success. When planning for variety, the planner must be able to find both short, successful plans and long, less-successful plans. Both PGraphplan and TGraphplan rely on incrementally increasing the time horizon of the planning graph, which prevents the planner from ever reaching the longer, less successful plans. In order to obtain the desired variety of results for this project, it is also required that the probabilistic planner used is able to plan for a given success probability and consider the time horizon of the plan and the cost of actions separately. These characteristics rule out Graphplan based planners.

In addition to direct extensions of the Graphplan algorithm, relaxed planning graphs have been used as heuristics for evaluating plans in other algorithms. One such algorithm, Probabilistic-FF (Domshlak & Hoffmann, 2006), has been very successful in planning competitions. Unlike other planners mentioned here, PFF performs a search over possible Bayesian Networks (BN) states. States are generated using a forward search and contain states that are reachable from the initial state of the planning problem after some number of available actions are applied. However, performing inference algorithms on BNs is #P-complete, and would impact performance in large problems. To solve this problem, the BN is first converted into a Conjunctive Normal Form (CNF) problem, which can be performed in linear time. Once in CNF, the literals of the problem are assigned weights from the BN. From there, the planner performs weighted model counting to solve the satisfiability problem (SAT) and return the expected probability of the plan. If this value meets the desired threshold, the plan is returned. Otherwise, the BN is extended with any actions whose preconditions are satisfied by the current state. Finally, after new states have been generated, a heuristic value of each new state is calculated using a relaxed planning graph. The underlying search strategy is ideally a greedy search using enforced hill climbing. Given a partial plan, Probabilistic-FF looks for the best single refinement that improves the plan. This process is then repeated with the improved plan. This search strategy can result in dead ends (search states from which a solution plan cannot be reached), at which point the planner falls back on a full breadth-first search (Hoffmann & Nebel, 2001).

This planner has proven to be very successful on large domains, and is able to solve problems with 100 initial world states and generates plans with 15-20 actions. However, like many other modern planners, the clear cost versus heuristic score found in Buridan-style planners is lost. It is important to the goals of this project that cheap, low probability plans are not passed over during the planning process. Modern planners such as Probabilistic-FF are too efficient at finding the highest probability plan to be useful for planning for variety. Particularly, the greedy forward search means that the shortest sequence in terms of discrete time steps (and without regard to individual action cost) will potentially be favored. Without the non-trivial inclusion of variable action cost, Probabilistic-FF is not suitable for planning for variety.

Somewhat ironically, one of the most successful planners in the International Probabilistic Planning Competition (IPPC) is essentially a deterministic planner. The FF-Replan planner (Sungwook, Fern, & Givan, 2007) takes as input a probabilistic planning problem and extracts a deterministic version. The most successful method for doing so chooses a single outcome of a probabilistic action and uses it as the sole outcome of a deterministic action. Choosing either the most likely outcome or the outcome with the largest impact (i.e. the one with the most effect literals) can often be successful. However, as the authors point out, one can easily choose a domain for which this method will perform poorly. Instead, therefore, FF-Replan creates a separate deterministic action for each possible outcome of each probabilistic action.

Once the planning problem is converted into a deterministic representation, the planner is able to search for a deterministic plan. When a solution is found, the resulting plan is then executed using the original probabilistic actions from which the deterministic actions were derived. If an unexpected state is produced, i.e. the probabilistic version of the action produced a different effect than the chosen deterministic sub-action, the planner re-plans using the newly found state as the initial state. This process continues until the goal state is reached.

Despite winning IPPC in both 2004 and 2006, the authors point out that this is largely due to the lack of problems that exploit the weaknesses of a re-planner. Little and Thiebaux (2007) attempted to define a probabilistically interesting problem. Their definition includes the following characteristics:

- Multiple Goal Trajectories: There must be more than one way to satisfy the goal conditions of the planning problem. (They are actually more specific, but for these purposes, this is sufficient.)

- Mutual Exclusion: The problem must force the planner to make choices to avoid actions that decrease the potential probability of success or result in a dead end.

Both of these properties are central to planning for variety. By definition, an agent must be able to accomplish its goals in multiple ways. Further, since agents can take actions that result in low-difficulty scenarios for the player, planning problems must contain actions that are low probability. Moreover, the planner must be able to purposely choose bad actions when finding low probability plans and choose good actions when needed to generate high probability plans. Despite their state-of-the-art performance in many domains, a re-planner is clearly not a viable option for finding a variety of plans.

2.2.4.1 Informative Heuristic using a Planning Graph

The largest impact on the performance of Buridan is the lack of an informative heuristic for ranking plans during the search. Two notable planners use the Buridan planner as a starting point and add plan ranking heuristics based on relaxed planning graphs. Rasmakrishnan *et. all* (2004) suggest using a planning graph to estimate the expected probability of incomplete plans and apply it to their planner PVHPOP (Probablistic Versatile Heuristic Partial Order Planner). Probapop (Nilufer, Welan, & Li, 2006) uses a relaxed planning graph to compute a distance analysis in terms of actions. While successful in planning competitions, Probapop uses a search strategy that is incompatible with maximizing plan variety. Once a complete partial order plan is found, the rest of the search queue is discarded. All subsequent improvements are made to this complete plan. This preempts the planner from backing out to refine other incomplete plans. While support for multiple search queues is planned and this is not an issue for some of the problems in the planning competitions, it is for the problems presented here (See Section 6).

In PVHPOP, like in Probapop and Probabilistic-FF, a relaxed planning graph is constructed with alternating layers of actions and propositions. Layers are added until all goal propositions are present in the final layer. Since multiple actions could potentially produce a proposition, *p,* the aggregate probability of *p* occurring is calculated by totaling the probabilities of any actions that produce *p* and subtracting the probability of both of those actions occurring. Let *A1* and *A2* be actions which produce *p* with probabilities of p_{A1} and p_{A2}, respectively.

$$\texttt{agg}(\texttt{p}) = \texttt{p}_{\texttt{A1}} + \texttt{p}_{\texttt{A2}} - (\texttt{Prob}(\texttt{A1}) * \texttt{Prob}(\texttt{A2}))$$

In this way, each proposition in the planning graph is assigned an aggregate value. This graph is constructed before the planning process, and the resulting information can simply be looked up during runtime. During the plan search, PVHPOP uses essentially the same refine-assess process as Buridan (see section 4.2). However, plans on the open list are ranked using a new process involving the planning graph:

1. Let *O* be the set of current open preconditions in the incomplete plan.
2. Assign each precondition an initial probability by looking up the aggregate probability calculated by the planning graph.
3. Project *O* through the incomplete plan by following the causal links in the plan. The probability of a proposition *p* after action *A* is given by *Prob*(*p*) = $A_{pre} * A_{eff}$, where A_{pre} is the probability of *A*'s preconditions occurring and A_{eff} is the probability of *A* producing *p*.
4. The final probability of all of the goal propositions occurring is then given by:

$$\texttt{Prob}(\texttt{P}_{\texttt{g}}) = \texttt{Prob}(\texttt{P}_{\texttt{g1}} * \texttt{P}_{\texttt{g2}} * \ldots * \texttt{P}_{\texttt{gN}}).$$

The resulting probability estimate is used in tandem with the cost of the plan to decide its rank. The empirically most successful Additive method defines the rank as:

```
Rank(plan) = k * Prob(plan) - Cost(plan)
```

The cost of a plan is simply a sum of the costs in all of the actions in the plan and *k* is a user defined weight.

In this way PVHPOP retains the easily adjustable and readable format of Buridan, while adding important optimizations using benefits provided by a relaxed planning graph. Unlike Proabilistic FF, PVHPOP does not rely on a greedy search algorithm. While not implemented here, this could potentially provide a much needed performance boost to the planner while retaining the necessary refine-assess process and easily adjustable heuristic found in Buridan. By weighting cost of partial plans more heavily than the probability estimate the planner should still be able to maximize the variety of possible solution plans.

3 Motivation

The need for probabilistic planning originates in robotics. Robotics applications often rely on sensors to build a picture of the surrounding world. Since these sensors often collect noisy data, it is important to take into account the uncertainty of the data. Probabilistic planners attempt to build plans that take this uncertainty into account. Video games do not usually have to rely on sensors. Rather, they often have the ability to simply “look up” information about the agents' immediate surroundings. Still, there are interesting ways in which uncertainty can be used to create better agents. By considering the way actions contribute to the overall success of a plan, the flexibility and quality of agents' behavior can be improved.

Another key difference between video game AI and a real-world domain like robotics is that the desired result is not necessarily the most successful. In order for players to feel engaged in a game experience, game designers must make the game difficult enough for players to feel a sense of accomplishment when they succeed, but not so difficult that they are unable to progress through the game. The properties of probabilistic planning can be leveraged to allow agents to dynamically solve problems in a way such that their resulting behavior creates a situation of the desired difficulty. This goal requires that the planner be able to find plans that have a variety of expected success rates.

GOAP was successful in F.E.A.R because it gave designers an intuitive set of tools that enabled them to create complex agent behavior. The planner resulting from this project should be able work with the same types of tools that were useful in GOAP. Additionally, controlling the plans found by the probabilistic planner should be equally straightforward to the designer. The problem definition in Buridan allows the user to specify a single value to notify the planner of the desired success threshold. Additionally, if more control is needed, the probability distributions of actions' effects could be adjusted. With the correct modifications, the planner can achieve a variety of plans by adjusting a single value, which could be done in real time, potentially adjusting to player performance.

Choosing a planner to be the starting point for this project depends on features that are not normally considered in the field of planning. The planner must go out of its way to evaluate plans that most planners are designed to avoid. This impacts efficiency and requires that it is possible to change the search strategy used by the planner. While the Buridan planner is orders of magnitude slower than modern planners, its structure is ideal for finding a variety of plans. POP, or SNLP (Systematic Non-Linear Planning), planners in general have largely been abandoned for applications which require sheer speed. However, for many applications, including the Mars rover Europa, the additional flexibility was necessary (Weld, 2010). While the requirements here are certainly less complex, planning for variety demands that the plans structure is not directly linked to a strict time frame and set action cost. Additionally, a video game environment allows us to consider a small controlled set of circumstances when compared with a real world environment. Domain specific modifications and different heuristics will be used to improve performance and the potential application of the heuristic used in the PVHPOP planner (See the Future Work section) will be discussed. While the latter is currently not implemented, the modified Buridan algorithm can generate an interesting variety of plans for game-like scenarios.

4 UCPOP and Buridan

In order to best present the Buridan algorithm, it is useful to first describe the partial order planner on which it is based. The Universal Conditional Partial Order Planner (UCPOP)

(Penberthy & Weld, 1992) is a least commitment partial order planner that uses a best first search algorithm to find a partially ordered sequence of actions that solve a problem. This algorithm is proven to be both sound and complete. However, the heuristic discussed below is inadmissible. This means that the heuristic function may overestimate the cost of refining an incomplete plan to a solution plan. While this potentially results in a sub-optimal solution, these cases have been have found to be rare in empirical tests and can essentially be eliminated by adjusting the strength of the heuristic.

4.1 UCPOP Algorithm Overview

The planner starts with an initial world state, a set of goal conditions, and a set of available actions. A solution to a partial order planning problem is defined as a series of actions that will transform the world state from an initial state to a goal state. These states are defined in the 'Start' and 'Finish' actions. As the planner runs, a list of open preconditions, *P*, is maintained. Initially, *P* contains the preconditions of the 'Finish' action.

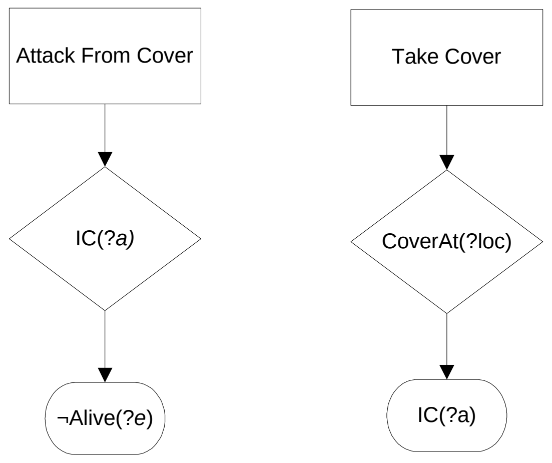

Figure 3: Example actions that could be used in a POP problem.
InCover(?agent) (IC) and CoverAt(?location)must be satisfied for the respective actions to produce the effects ¬Alive(?enemy) and InCover(?agent).

The planner then begins to refine the plan. In each refinement step, the planner chooses either an open precondition or an open threat. Let p be an open precondition of some action A_j. For each action A_i, which can support p, a new child plan is generated. Each child plan has several changes from its parent. First, the causal link, A_i--p--A_j, is added to record which action has satisfied the precondition. Second, an ordering constraint is added to ensure that the action producing p occurs before the action consuming p. Finally, any preconditions of A_i are added to the list of open preconditions. If a designer wants an agent to attempt to kill an enemy, he or she might use the actions in Figure 3. Here the goal state, P_g, is set to be **¬Alive**(*enemy*). The planner would support P_g with the **AttackFromCover** action and adopt the new precondition **InCover**(*agent*). The planner could then use the action **TakeCover** to support the new precondition. Finally, assuming there is at least one cover location defined in the initial state, the **CoverAt**(*location*) precondition would be supported and the complete plan would be returned.

However, problems are not always so straightforward. There are times, like the Sussman Anomaly, where actions will not be strictly ordered in a convenient way. Since actions are not strictly ordered by default unless connected by a Causal Link, the planner must detect possible threats caused by unordered actions. An Action *At* is said to threaten a Causal link, A_i-*p*-A_j, if it has an effect ¬*p* and can possibly occur in-between A_i and A_j. In this case, the action A_t must be strictly ordered to occur either before A_i (demotion) or after A_j (promotion). When using universally bound variables, as discussed below, the planner can also employ a technique called separation. Separation is used when a threat involves one or more literals with unbound variables, and resolves the threat by adding binding constraints such that future refinements of the plan cannot bind the literals' variables to invalid world objects.

When checking for threats in a plan, the planner must determine which actions could possibly occur within a causal link. To do this, it is useful to compute a partial ordering of the plan. A partial ordering of a plan assigns a time index to each action. The time index represents the actions' temporal relation to one another. If action A has a lower time index than B, it is guaranteed that A will execute and complete before B is executed. However, as the name implies, a partial ordering may have actions with the same time index. A set of actions with the same time index may be executed in any permutation of orderings. If the actions A, B and C all have the same time index, it must be assumed that they could be executed as ABC, ACB, BAC, BCA, CAB, or CBA.

While the planner could consider any action not strictly ordered outside the link to be a potential threat, this increases the size of an already large search. By using the partial ordering of a plan to identify threats, the potential threats are limited to those which can actually occur during a causal link, even if they are not explicitly ordered to be outside it. For example, there may be an action *A* with a time index T_a, and a causal link with actions B and C (which implies the indices T_b and T_c such that $T_b < T_c$). Only if $T_a >= T_b$ and $T_a <= T_c$, must the planner evaluate action A as a threat to the causal link.

In order to determine the partial ordering of a plan, the planner must perform a topographical sort of the actions. Using the ordering constraints as edges of a graph, the maximum distance of each action from the 'Start' action can be determined. This distance describes the time index in the plan's execution at which the action needs to be executed. Actions with the same index may be executed in parallel or in any permutation of strict orderings.

4.1.1 Action Variables

In order to maximize expressivity, UCPOP allows actions with universally bound variables. This allows a single action to be used in multiple ways. In "Block World" the goal is to be able to move any of the three blocks with the same **Move**(*?block, ?from, ?to*) action, rather than create the specialized actions like **MoveAFromXtoY**, **MoveBFromZtoX**, etc. Like the temporal ordering of actions, the planner uses a least-commitment methodology when determining to what a variable is bound. When the planner adds a new causal link to a plan, only the variables relevant to the new link are bound. Using this technique allows the planner to backtrack and try different bindings if the planner reaches a dead end.

In order to determine if universally bound literals are compatible, the planner must find the most general unifier between two literals. Penberthy et. all (1992) define the function MGU (Most General Unifier) to handle this problem. Its results can best be described in three main classifications. Here, it is assumed that the literal type is always the same (i.e. **Clear**(*?a*) and **On**(*?b,?c*) will not be compared, even though *?a* could potentially match both *?b* and *?c*).

- Fully Bound

  Example: MGU(**Clear**(*A*), **Clear**(*A*)) => {}

  These literals are compatible and require no additional bindings to be equivalent.
- Compatible but not fully bound

  Example: MGU(**Clear**(*?a*), **Clear**(*A*)) => {(*?a, A*)}

  These literals are compatible and will be equivalent if *?a* is bound to the world object A.
- Incompatible

  Example: MGU(**On**(*?a, B*), **On**(*?a, C*)) = $\perp$

  There is no way in which these literals can be made to be equivalent

Once it is determined that two literals are compatible using the above rules, they must also be checked against the global bindings present in the plan. As mentioned above, the separation threat resolution technique adds a binding constraint that prevents the literals in the threatening action from being bound to same value as those in the causal link in future refinements of the plan. The planner must take these constraints into account whenever two literals are checked for compatibility.

Similar complexities arise when looking for actions to support an open precondition. There are three cases the planner must consider. When neither the open precondition nor the action effect literal has bound variables, the planner binds them to be equal. Once they are bound to concrete values, the values can be propagated to one another. This becomes more complex when the variables are partially bound. In this case, the planner must ensure that any bound variables are bound to the same value. Similar to before, if a variable is unbound in both, the planner binds it to be equal. If all variables are bound in each, the planner only must ensure that each is bound to the same thing. In each case, if any single variable is bound to be different in the two literals, they are deemed to be incompatible.

The way the planner identifies and resolves threats must also be changed. Compatibility between literals is handled the same way as described above. However, here the planner must check whether a literal could possibly negate the literal in a causal link. If the two literals in question have variables which match, or could possibly match in future refinements of the plan, a new threat is added to the plan. To resolve the threat the planner can still use promotion or demotion. However, these techniques are not always possible, as an action could be strictly ordered to occur within the causal link. In this case the planner relies on the aforementioned technique, separation. This technique involves binding the unbound variables in a threatening literal so that they cannot interfere with the causal link. Specifically, each variable is bound to be not equal to the corresponding variable in the causal link.

4.1.2 Plan Validity

After supporting a causal link with an existing action or resolving a threat, the planner must check the plan for cycles and binding conflicts. To detect cycles, the planner again uses Tarjan's Algorithm (Tarjan, R.). Here, it is modified slightly to check for nodes already visited and treat the actions in the plans as vertices and the ordering constraints as edges. After doing this and choosing a start node, a rooted, directed graph is passed to the algorithm. The planner must run the algorithm with each action as the start node. If a cycle is found from any point in the plan, the plan is deemed invalid and is not added to the open list.

The planner also must ensure that each plan has consistent bindings. Specifically, a valid plan does not allow a variable to be bound to two different concrete values. Nor does a valid plan allow a variable to be bound to a value from which it is separated. This can mostly be handled when the planner is supporting a precondition and resolving a threat by separation. However, at times, a value may be propagated through two or more actions when a new causal link is added, so the planner must also check during the propagation process. Again, an inconsistent plan is not added to the open list.

4.1.3 Plan Search

The search tree for UCPOP potentially has a very high branching factor. Consider a refinement in which a precondition can be solved by many different actions, which all produce the desired effect. The planner must generate a child plan for each possible solution to the open precondition. Similarly, there are up to three branches if the plan has a threat. The algorithm is exponential, but can effectively take advantage of a best first search.

The most basic aspect of a plan that is valuable as a heuristic is the number of actions. Ideally, the plan should be as efficient as possible, so the planner should look at plans with the fewest actions first. When only using this as its guide, the planner performs a breadth-first search. Within the available plans with the same number of actions, the planner should favor plans that are closest to a solution. There are several different heuristic options to choose from. The most promising of these favor plans with the fewest preconditions, the fewest threats, the most causal links, or a combination. In general, favoring plans with the lowest rank, r, where $r = |A| + |P| + |T|$ yields the best results. Here, A is the set of actions in the plan, P is the set of open preconditions, and T is the set of threats.

The planner also needs to be able to backtrack if a series of refinements reaches a dead end. In UCPOP, a dead end occurs when there are no longer any valid refinements and the plan is incomplete. At this point, the planner looks at the next plan in the open list without adding any new child plans. Or, if the open list is empty, there is no valid solution to the problem.

When compared with modern planners, the heuristic used in UCPOP is not very informative. That is to say, simply looking at the number of open preconditions and threats, or any other basic features of a partial plan for that matter, does not consistently point towards a solution. A partial plan with a single open precondition and no threats may still be many actions away from a solution, which could require many more plan refinements than expected. Likewise, a plan with a few open preconditions and/or threats could be very close to a solution, but would get stuck low in the open list until many other plans were examined first.

4.2 Buridan

Buridan (Weld, Hanks, & Kushmerick, 1994) is a probabilistic planner, and an extension of UCPOP. Most real world environments, and often those in video games, have some degree of uncertainty. This could be uncertainty about what will happen in the near future or uncertainty about the current state of the world. There are several changes in the Buridan algorithm from UCPOP, but the most important is that Buridan searches for a plan which achieves a provided success probability. Even if a complete plan (a plan in which all of the preconditions are met) is found, the planner will continue to reinforce the plan until it meets the threshold. In order to take these factors into account, changes must be made to the way the world state and actions are represented.

4.2.1 Plan Representation

Instead of a single set of preconditions and effects, actions are now made up of data structures called “consequences”. Each consequence is made up of a trigger *t*, a probability *p*, and a set of effects *e*. A trigger is a Boolean expression of world state literals, similar to a set of preconditions. For simplicity, the trigger will be referred to as a set of preconditions from now on. When the preconditions for a consequence are satisfied by the world state, the effects have some likelihood, given by *p*, of actually changing the world state. For example, if a game designer wants an agent's ability to successfully attack to depend on whether or not the agent and enemy are in cover, he or she might use the action in Figure 4. In each case, the agent has some chance of being successful. To achieve the goal **¬Alive**(*enemy*), a plan could contain only the attack action. However, to achieve a success probability that is greater than .5, the agent will need to attempt to find cover and flush the enemy from their cover in some way.

This action representation change also affects the initial world state. Instead of having a single set of literals defining the world state, the 'Start' action can now have any number of consequences describing possible world states and their respective probabilities. The coupling of actions that may or may not produce the desired effects and multiple world states produces the necessary structure to represent a world while considering uncertainty.

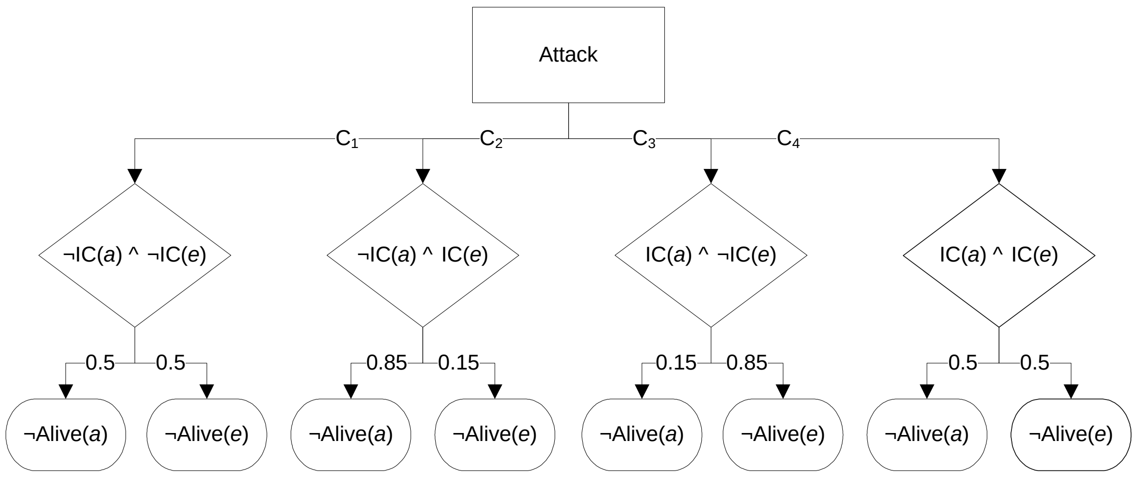

Figure 4: Example of an action that could be used in Buridan
The success of the agent depends on whether the agent(a) and enemy(e) are in cover (IC). Successful agents will attempt to find cover and flush the enemy from cover.

4.2.2 Algorithm Overview

The refinement step in Buridan is generally similar to UCPOP, but there are some important differences. First, Buridan allows a precondition to be supported by multiple actions or multiple consequences of the same action. In practice, the planner still looks at unsupported preconditions first and, once a complete plan is found, attempts to improve the plan by reinforcing all of the preconditions of the plan. This will be discussed further below.

Buridan also requires another threat resolution technique. Since actions can have multiple outcomes, the planner must allow for the case where a threat exists, but the plan is still sufficiently likely to succeed. The Confrontation resolution technique searches for a consequence which does not threaten the link. If one is found, it is marked as safe to the threatened link. Additionally, the preconditions of the safe consequence are added to the open preconditions. In this way, the planner is pointed towards making the safe consequence the most likely to occur.

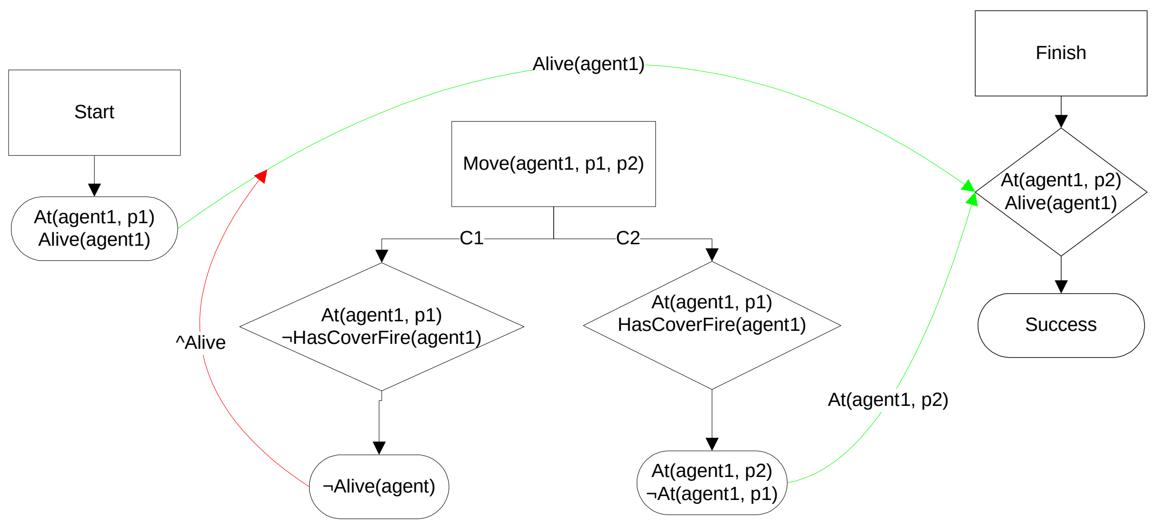

Figure 5: Resolving threats by confrontation

The first consequence of Move threatens the Alive(agent1) causal link, and has been denoted with the link, since the second consequence is safe.

In the example shown in Figure 5, the first consequence of the **Move** action threatens the causal link from **Start** to **Finish**, which supports the literal **Alive**(*agent1*). However, the second consequence does not threaten the link, so the planner can mark the first consequence as safe to the causal link. This allows the planer to ignore the threat in future refinements. Additionally, the planner adds the precondition, **HasCoverFire**(*agent1*) to the list of open preconditions. In this way, resolving threats by confrontation points the refinement process in the right direction.

4.2.3 Assessment

To determine the success probability of the plan, it must be assessed. The simplest assessment method is called Forward, which calculates a lower bound of success probability for a plan. To accomplish this, the planner must first find all of the total orderings for a plan. In the case where two or more actions are not strictly ordered before or after one another, the planner creates a total ordering for each permutation. Each initial world state is then projected through each total ordering. When a world state is projected through an action, the triggers of each consequence are evaluated. Once a match is found, new world states are created for each outcome of the consequence. The state's probability is updated: $P_s' = P_s*P_c$ and the world state is updated with the consequences effects. Once the planner has projected the world state through each action in a total ordering, the probability of each world state that satisfies the goal condition is accumulated, representing the probability of success of *this* total ordering. Once the planner has probability values for each possible total ordering, the lowest is chosen, guaranteeing the lower bound.

4.2.4 Search Control

The heuristic for Buridan is the same as in UCPOP. Surprisingly, there is no consideration given to the assessed probability of plans. This means that once Buridan reaches the stage in which it is supporting all of the preconditions in the plan (referred to here as the Buridan stage), there is no difference in rank between a complete plan that has been partially reinforced (a new action has been added but its preconditions have not yet been supported) and a plan with the same number of open preconditions that has not yet reached that stage. For simple problems, like the examples provided by (Weld, Hanks, & Kushmerick, 1994) this problem is not apparent. The 'SlipperyGripper' problem, for instance, requires that a robot's gripper is dried before a robot picks up a block. In this problem, the **DryGripper** action does not have any preconditions or threatening effects, so when the action is added, the newly generated plans are put on the top of the open list.

In even slightly more complex problems, where a sequence of one or two actions with preconditions or effects require further refinement steps, plans that are closest to a solution are potentially relegated to the bottom of the open list. Additionally, during the Buridan stage, *all* preconditions in the plan are reinforced. The result is a very large branching factor, which is given by $(A_1 + A_2 + \ldots + A_N)$ where A_n is the number of ways in which precondition *n* could be supported and *N* is the number of preconditions. It should be noted that this includes applicable existing actions in the plan and that actions can potentially have multiple consequences that support a literal. Each of these refinements generates a new child plan, resulting in a highly polluted open list. Without an effective way to rank partial plans, the end result is an intractable search for the large majority of non-trivial problems.

This poor performance was confirmed by Blum *et. all.* (1999) when they tested Buridan with a problem similar to SlipperyGripper, called "Probabilistic Blocks," which involves a robot inverting a stack of blocks. With two blocks, Buridan was able to solve the problem in a reasonable amount of time, resulting in a plan with four actions (not including the initial and goal actions). However, when tasked with the same problem using three blocks, which would require a solution plan with eight actions, the planner was unable to find a solution within 100,000 plan refinements. As will be seen in the Two Agent Cover Advance Problem example presented in section 6.1.2, once plans reach a certain critical combination of size and complexity, the size of the search space becomes prohibitively large.

4.3 Planning for Variety with Buridan

In order to apply this algorithm to a video game environment, the algorithm must be altered in several ways. While the standard Buridan is written to be as general as possible and strictly adheres to a bare-bones problem description format, this project has the advantage of applying the algorithm to a fairly specific domain. This both allows for some convenient assumptions and requires additional features. Additionally, it is useful discuss how commonly used AI information can be represented with Buridan-style action definitions.

4.3.1 Heuristic

Since the main goal of this project is to get a variety of plans depending on the given threshold of success, the planner should examine low probability plans before high probability plans. To accomplish this, a cost is added to each action, which is then used in place of plan size when calculating the rank. This allows a designer to create situations where a cheaper action is less likely to succeed and an expensive action more likely. In this way, the cost of the plan is separated from the number of actions.

In many cases it is useful to consider the probability of plans which have reached the Burdian stage. By including probability in the heuristic, the planner can attempt to keep these plans at the top of the open list. Once a plan reaches the Buridan stage, its success probability is stored and propagated to child plans. Since children of the plan will likely have new open preconditions with unresolved variables, the planner does not need to re-assess its children plans until they are refined back into a complete plan. The cost part of the ranking is still most important when ranking plans which have a success probability of 0.0, so cheaper incomplete plans will be refined to the Buridan stage first.

However, it is possible to design a problem such that the planner will skip possible solutions when using this heuristic. Let *A* and *B* be two complete plans (no open preconditions or threats) which are solutions to a problem, but do not meet the success threshold. Also, let *a* and *b* be the cheapest set of actions required to refine each plan into a solution which meets the success threshold. *A* is a cheaper plan, and will reach the Buridan stage first. After some number of refinements, the set of actions, *a*, will be added such that *A* meets the desired threshold of success. If $Cost(B \cup b)$ is less than $Cost(A \cup a)$, the planner will have skipped a cheaper solution which still would have met the desired threshold of success. However, in the domain of video games this would be an odd design choice, and would be contrary to the proposed use of this system. When designing example problems, it became apparent that interesting problems have one or more primary sequences of actions (i.e. actions which are required to form a plan with a non-zero probability of success) where cheaper sequences are less likely to succeed. Secondary actions, or sequences of actions, can then be added once the plan reaches the Buridan stage. Here, the assumption (using the above example) that if $Cost(A) < Cost(B)$, then $Cost(A \cup a) < Cost(B \cup b)$ is added.

4.3.2 Actions with Variables

Another important distinction between the implementation used here and the one presented by (Weld, Hanks, & Kushmerick, 1994) is the addition of actions with variables. This is a feature present in UCPOP and is a proposed feature for Buridan. The propositional representation used in the standard Buridan algorithm requires a distinct action for each object in the world. For example, to move an agent to two different locations, the agent needs the actions **MoveAgentToPos1** and **MoveAgentToPos2**, which would have the effects **AgentAtPos1** and **AgentAtPos2**. When applying Buridan to a game-like situation, it is desirable to be able to change the initial world state and goal expression without having to add actions that specifically cater to the current circumstance.

By allowing unbound variables, designers are able to create actions like **Move**(*?agent, ?location*), where the variables *?agent* and *?location* could be bound to any agent or location described in the world state. This allows them to reuse the same action for any agent moving to any location. This action produces the effect **At***(?agent, ?location)*. When the action is used to support a precondition which contains concrete literals, **At***(soldier, position1)* for example, the variables in the new **Move** action are bound to the new concrete literals. Additionally, the bindings are propagated to any actions that are connected via causal links. If, at some point, bindings are in conflict, the plan is deemed invalid and is not added to the open list.

4.3.3 Domain Specific Assumptions

Efficiency is also an important consideration when building games. As mentioned before, the Buridan planner performs extremely badly on non-trivial problems. By giving the planner additional information about the problem, however, the size of the search can be reduced considerably. The most effective ways of doing this is marking which preconditions are allowed to be supported more than once.

When testing likely game scenarios with Buridan, it becomes clear that the designer needs to have a plan in mind that will achieve that maximum success probability. Consequently, the designer of the actions will know which preconditions can be supported effectively. For instance, if the planner is trying to satisfy the precondition **At**(*location1*), it might add the action **Move**(*agent, location1*). In this case the agent will either make it or not make it. Adding another **Move** action will not raise the probability of the plan, so it is only useful to support this precondition once. Conversely, if the planner tries to support the precondition **Covered**(*agent*), it could first add a **RequestCoverFire**(*agent2*) action and then increase the probability of the plan by adding a **ThrowGrenade**(*enemy*) action.

Similarly, the designer will know how many actions will be in the largest possible plan. By limiting the planner to this many actions, the planner can avoid examining many unnecessary plans. Favoring low cost plans also helps accomplish this. However, the planner must consider the case where the only plan which can meet the threshold of success has a very expensive action. This would cause the planner to search through many large plans made up of cheap actions until the total cost becomes greater than that of the solution plan. Limiting plan size allows us to avoid known dead ends.

4.3.4 Representing AI World Information with Buridan

In order for Buridan to be usefully applied to game environments, it must be able to utilize existing common AI world state information. When using FSMs, the agent must be able to access information about the world in order to know when to change states. Planners can take advantage of the same information and can more effectively take advantage of some of the more complex types of information.

Most types of world information are easily converted to Boolean literals that are easily usable by the planner. The examples presented here require:

- Locations (agents and cover).
- Agent state (In cover, covering another agent, being covered by another agent, current weaponry).
- Relationships (distances between locations, visibility between locations).

Combined, these three simple types of information can be used to represent more complex situations. For an agent to move around safely or effectively attack an enemy, it must be aware of which locations are in and out of range relative to enemy agents or the player. In practice, it is important that a plan have actions that check if a particular agent is in range of a target object. An **Attack** action for example may require the agent to be in range of an enemy. However, since the agent and/or enemy may move around during plan execution, it is not useful to explicitly declare a literal **InRange**(*agent, enemy*). Instead, the combination **InRange**(*loc1, loc2*) ^ **At**(*agent, loc1*) **&** **At**(*enemy, loc2*) is used. By declaring a set of **InRange** literals, the planner can reason about the best way to proceed using a simple representation of a game environment, similar to a visibility graph.

5 Planning Example

Here a concrete example is presented that could be used in most action games. In the initial state, an agent has entered a combat area and is in the open, possesses a gun and grenades, and is faced with an enemy that is in cover. There is also a cover location close to the agent and both the agent and enemy are alive. The agent's goal in the example is to kill the enemy and stay alive with a probability of 0.7.

The agent is also assigned a set of actions. In this example, the agent can take cover, throw a grenade, and attack with a gun either with or without cover. Constraints are also added to the plan. Here, a maximum of 5 actions (including **Start** and **Finish**) are allowed. The designer-provided preconditions are set to be **¬Alive**(*enemy*) for **Finish**, **¬InCover**(*enemy*) for **Attack**, and

InCover(*agent*) and **¬InCover**(*enemy*) for **AttackFromCover.** In this example, each precondition can only be supported once.

The planner starts with the empty plan, consisting of the initial state and the goal expression, represented by the **Start** and **Finish** actions. The first precondition it tries to satisfy is **¬Alive**(*enemy*) and generates two child plans using the actions **Attack** and **AttackFromCover**. The **Attack** action is cheaper, so the planner refines the plan with this action first. Each of the preconditions in *consequence2* of **Attack** is met by the initial world state, so the first complete plan the planner encounters has a cost of 1 and the probability of success is 0.15.

Because the first complete plan does not meet the desired probability threshold of 0.7, the planner must further refine the plan. This time it chooses the designer-provided precondition **¬InCover**(*enemy*) of **Attack**. The **ThrowGrenade** action is added to the plan (Figure 6), which makes the the new plan have a total cost of 3 and probability 0.43. This plan still does not meet the probability threshold, and each of the designer provided preconditions is satisfied to its limit, so the planner must backtrack.

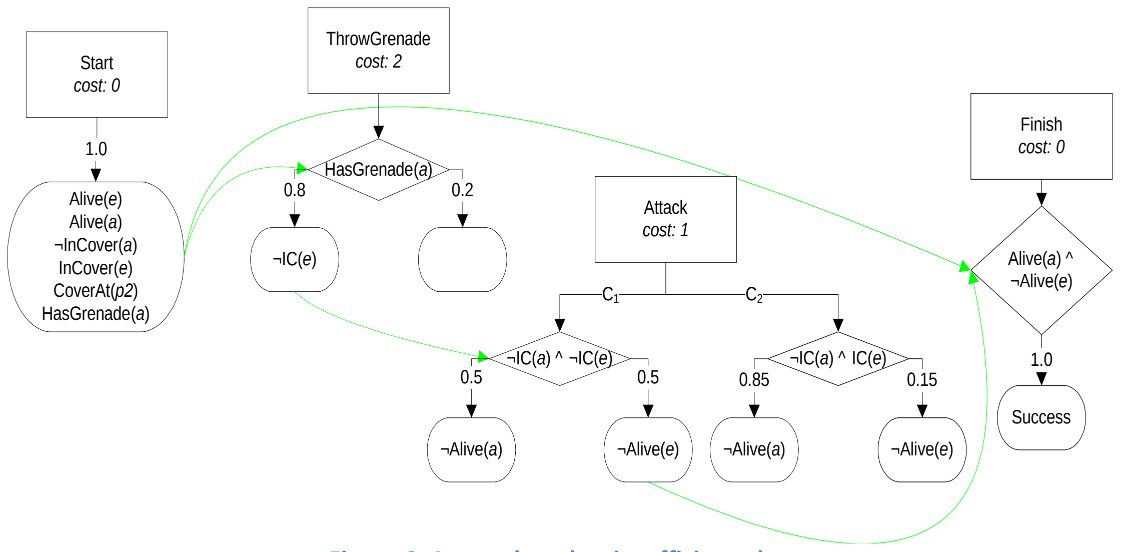

Figure 6: A complete, but insufficient plan

A plan with a success probability of .43, as found by performing the forward assessment algorithm.

The first plan on the open list is the plan with the **AttackFromCover** action that the planner generated above. The planner refines this plan by supporting **InCover**(*agent*) with the **TakeCover** action. Once again the plan is below the threshold, with a probability of .48. This time, however, the planner can further refine the plan by supporting the precondition **¬InCover**(*enemy*) by adding the **ThrowGrenade** action. This plan (Figure 7) has a cost of 4 and a success probability of .78, which exceeds the threshold of success. The solution plan with the actions **ThrowGrenade, TakeCover,** and **Attack**, is returned and executed.

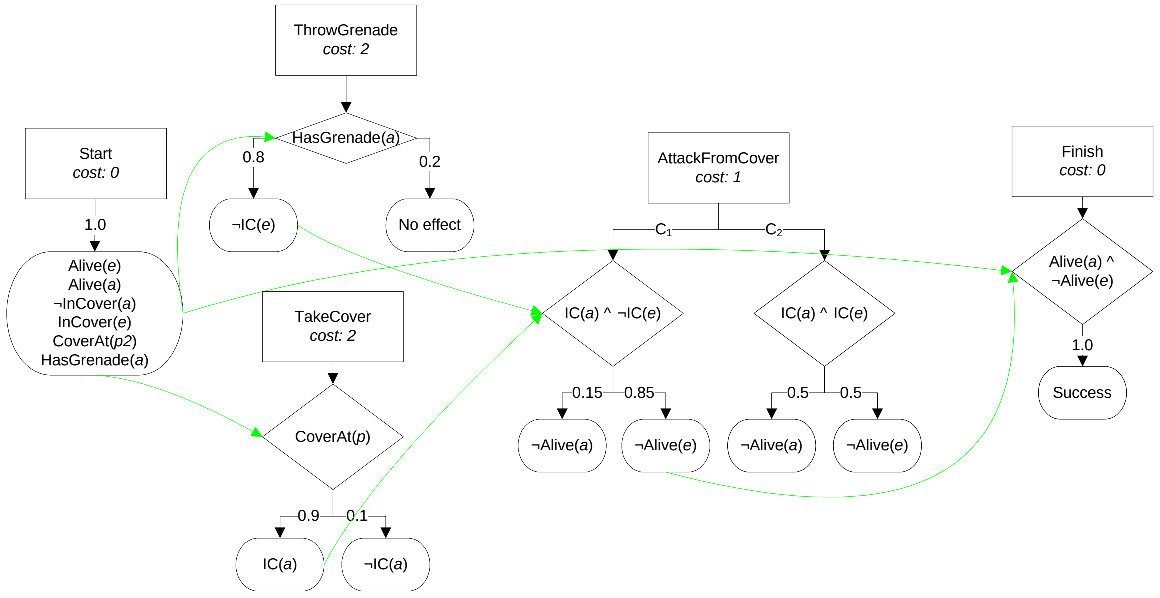

Figure 7: Planning example solution

The solution plan with a success probability of .78.

If the support limit of the precondition **¬InCover** was increased by one, an additional **ThrowGrenade** action would be added to increase the probability of the plan. This is because the action does not produce any negative effects when it fails. Conversely, an attempt of the same strategy with the **InCover** precondition and the **TakeCover** action would not increase the probability of the plan since, if it doesn't succeed, it produces the effect **¬InCover**. These types of properties can be employed to generate plans with the desired properties.

Adjusting the cost of actions can also play a large part in which plans are found, and in which order. While probability is more heavily weighted, it has no effect when all of the plans have a probability of zero. In this stage, the planner will refine cheaper plans first. Then, if the threshold is not met, the planner backtracks and moves on to more expensive options. The most important result of this is that the probability of the solution tends to stay close to the goal probability, maximizing the variety of solutions given different probabilities.

6 Demo and Results

Buridan, with the modifications presented here, can successfully solve game-like problems. This allows an agent's level of difficulty to be modified by simply changing the success threshold. Additionally, other AI modules could be used to control both the probabilities in actions as well as the overall success probability. If a game were to keep a record of common player strengths and weaknesses, the actions probabilities could be adjusted dynamically as the game progresses. If the player is dying too often, or not enough, the success probability could be lowered or raised. With well designed actions, this would cause the agents to act differently and to make different decisions, rather than simply raising or lowering hit-points or accuracy, as is currently found in many games. While the size and complexity of the problems the planner is able to solve in a reasonable time is still limited by the lack of an informative heuristic, it is useful to examine the planner's ability to produce a variety of plans and both evaluate the modifications made to the Buridan algorithm and suggest promising avenues for future work.

6.1 Plan Variety and Performance

The primary goal of this project is to produce a variety of plans given a problem and a desired probability of success. The primary focus here will be on the planner's merits in this area, but efficiency and scalability will also be evaluated. Four primary examples were used in order to test the Buridan algorithm with the modifications described above. The features which are focused on in these problems include (1) interleaving of actions, (2) handling of multiple potential world states, (3) supporting preconditions with multiple actions, and (4) generating different plans based on the given success probability. All of the examples emulate the types of problems video game agents would likely encounter.

Performance is also of major importance to video games, so the planner will be evaluated with each of the modifications in each problem. Ultimately, planning time is the primary focus. However, the number of plans refined and the number of child plans generated will also be examined. As will be shown, fewer refinements do not always result in shorter planning times. Finally, the planner's ability to determine that there is no solution for a problem will be tested. Buridan is not guaranteed to deduce that there is no solution (Weld, Hanks, & Kushmerick, 1994), so it is important to evaluate the implications of this when applied to a video game environment. In practice, it would likely be sufficient to return the highest probability plan found within a certain threshold of CPU time, but it would be ideal to attain completely predictable outcomes.

Each problem is tested with the C++ implementation of the standard Buridan algorithm (which includes the addition of actions with variables), the addition of designer marked preconditions (DMP), the probability heuristic (PH), and the addition of a maximum action threshold (MA). Planning times are given in milliseconds, and are followed by the number of plans refined (r) and the total number of child plans generated (c).

6.1.1 Problem 1: Cover Advance

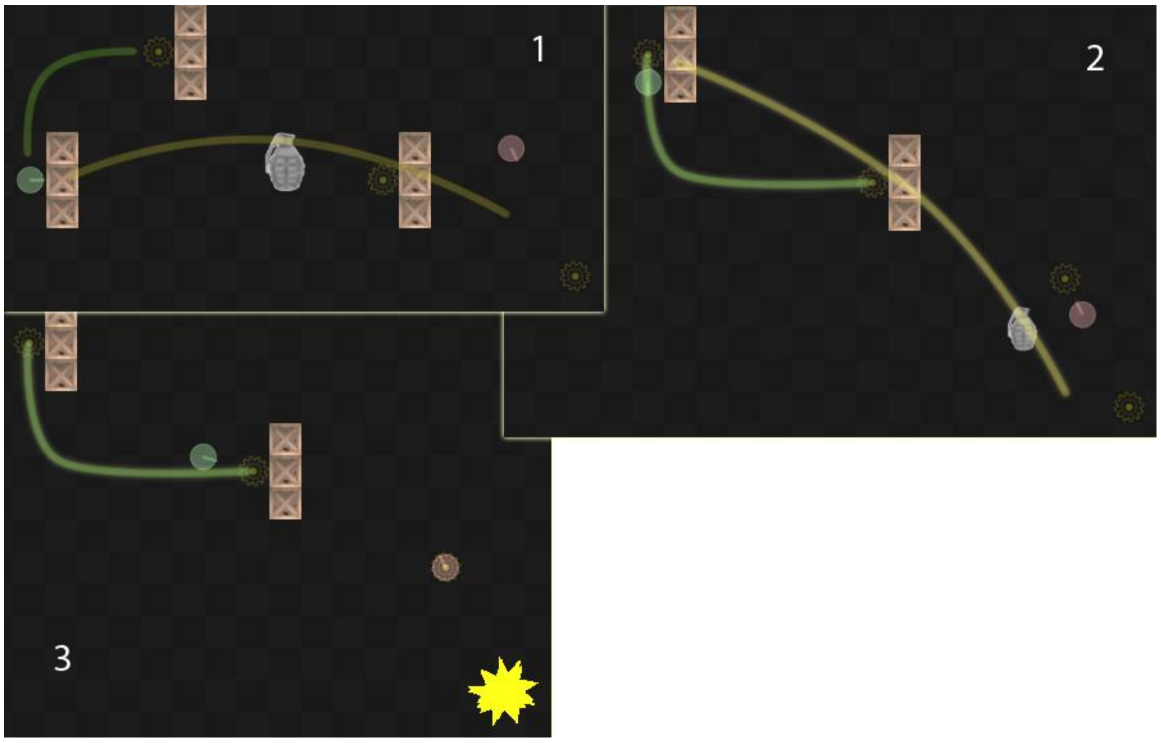

Figure 8: CoverAdvance demonstration sequence

The agent is more likely to survive the moves from cover to cover if it has thrown a grenade to distract the enemy.

The CoverAdvance problem involves a single agent who must advance from cover to cover to reach a forward position near an enemy. Longer advancing moves are considered to be lower probability, as the enemy will have more time to attack unobstructed. By throwing a grenade, the agent can distract the enemy and provide cover for itself. Distances are described using literals of the form **DistN**(*?loc1, ?loc2*), where N is some numerical distance. In this example, N is either 1 or 2. The actions Move1 and Move2 use the corresponding distance literals, **At**(*agent,?loc1*) ^ **Dist1**(*?loc1, ?loc2*), as preconditions. In this way, the plan can utilize a graph of distances between various locations to find the most effective way to proceed. For this example, higher probability plans will involve more grenades and shorter advancements (Figure 9). Here, *Pg* = **At**(*agent*, *p3*).

p = 0.5	P = 0.7	P = 0.76	P = 0.9
Move2(agent, p3)	ThrowGrenade	Move1(agent, p2)	ThrowGrenade
	Move2(agent, p3)	ThrowGrenade	Move1(agent, p2)
		Move1(agent, p3)	ThrowGrenade
			Move1(agent, p3)

Figure 9: Plans generated for the CoverAdvance problem.

This problem is significantly more difficult for the standard Buridan planner to solve at higher probability thresholds. The **Move1** action has four outcomes depending on whether or not the agent in question is covered, each of which has three preconditions. Once the planner reaches the Buridan stage, it must attempt to reinforce each precondition in each Move action, often resulting in a branching factor of more than 30. The results is in an extremely large open list and, moreover, each time a plan reaches the Buridan stage, the generated child plans are often inserted low in the list, as they have new open preconditions and additional actions.

CoverAdvance			
	Marked preconditions, Max actions, Prob. Heuristic	Marked preconditions, Max actions, POP first	Std. Buridan
probability	**time (ms) [r:(plans refined) c:(children generated)]**		
0.5	0.42 [r: 8 c: 17]	0.4 [r: 8 c: 17]	0.4 [r: 8 c: 17]
0.7	0.95 [r: 16 c: 35]	0.9 [r: 16 c: 35]	1.37 [r: 16 c: 47]
0.76	2.05 [r: 35 c: 63]	2.05 [r: 35 c: 63]	10.22 [r: 60 c: 208]
0.9	4.41 [r: 72 c: 113]	4.4 [r: 72 c: 113]	64.61 [r: 235 c: 651]

Figure 10: Performance results for the CoverAdvance problem.
The probability in the left column is the threshold of success a solution plan must meet.

By limiting the refinements at the Buridan stage to only those specified by the designer, the planner can greatly reduce the number of child plans that are generated and refined (Figure 10). In this case, including the probability of partial plans in the heuristic has no effect on the size of the search. Since the actions added to the plan for this problem can only be the **ThrowGrenade** action, child plans resulting from Buridan stage refinements only have a single open precondition. Here, ranking plans solely on open preconditions and threats happens to result in a convenient ordering. However, it is important to observe that, in this form of problem, utilizing the probability heuristic is not a detriment to performance, nor does it cause the planner skip over any of the possible solutions to the problem.

6.1.2 Problem 2: Cover Advance (two agents)

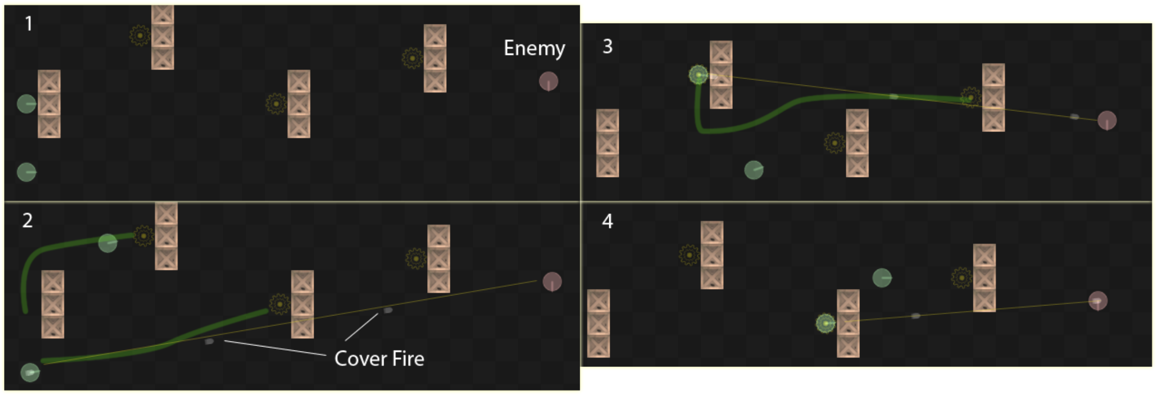

Figure 11: Two Agent CoverAdvance demonstration sequence
Agents are more successful if their teammate is providing cover fire for them.

This problem is similar to the Cover Advance problem, but with two agents. In order for one agent to provide cover fire for another agent, it must move to be in range. This results in agents which work together to progress forward. While Buridan is not a complete solution for squad-based planning, it can solve problems that concern a small number of agents. Thought of as a single character, these agents can work together to accomplish a goal. In this example, agents can **Move** from one cover location to another and provide cover fire for another agent. When moving, the agents' success probabilities are improved if another agent is providing cover fire. As in the above example, agents can only move to cover locations which are in range. Here, an agent can only move and provide cover from a distance of one. The goal in this problem is for agent1 to reach the location p5. Plans with a higher success probability result in a pattern where one agent moves while the other covers (Figure 12).

P = 0.25	**P = 0.49**	**P = 0.74**	**P = 0.98**
Move1(agent1, p3)	Cover(agent2, agent1)	Cover(agent2, agent1)	Cover(agent2, agent1)
Move1(agent1, p5)	Move1(agent1, p3)	Move1(agent1, p3)	Move1(agent1, p3)
	Move1(agent1, p5)	Move1(agent2, p4)	Cover(agent1, agent2)
		Cover(agent2, agent1)	Move1(agent2, p4)
		Move1(agent1, p5)	Cover(agent2, agent1)
			Move1(agent1, p5)

Figure 12: Plans generated for the Two Agent CoverAdvance problem.

Again, this problem proved more difficult for the standard Buridan algorithm. The problems encountered when solving the CoverAdvance problem are more costly here. In order to solve the problem at the success thresholds 0.74 and 0.98, the planner must add multiple actions after reaching the Buridan stage (Figure 13). In the standard Buridan algorithm, and to some extent in the modified planner without the probability heuristic, the result is an explosion in the size of the open list. Without the probability heuristic to keep the newly generated, promising new plans near the top of the open list, their higher cost results in a place low in the open list. The planner must then exhaust all possibilities of lower cost before making the desired refinements. Additionally, the standard Buridan algorithm is forced to examine large plans that are extremely expensive to assess. This is indicated by the fact that the planning time is approximately 20 times longer, but only 13 times as many plans have been refined when compared to the results of planning with marked preconditions and without the probability heuristic. Particularly, without limiting the number of times the **Covered**(*?agent*) precondition is supported, the planner generates plans similar to the P=0.49 plan in Figure 12, except with multiple **Cover** actions. Each of the cover actions has the same topographical ordering. Recall that when assessing with the *forward* algorithm, we must consider every possible total ordering. For instance, in the case with 7 cover actions, the planner assesses 7!, or 5040, plans. Since adding additional **Cover** actions does little to increase the success probability of the plan, the planner is essentially wasting time. In this problem, designer marked preconditions are crucial to both eliminate the generation of useless plans and to avoid costly assessment operations.

CoverAdvance 2 agent			
	Marked preconditions, Max actions, Prob. Heuristic	Marked preconditions, Max actions, POP first	Std. Buridan
probability	**time (ms) [r:(plans refined) c:(children generated)]**		
0.25	0.95 [r: 12 c: 22]	0.92 [r: 12 c: 22]	0.87 [r: 10 c: 19]
0.49	4.11 [r: 41 c: 83]	3.9 [r: 38 c: 76]	10.13 [r: 114 c: 205]
0.74	23.39 [r: 179 c: 299]	131 [r: 1048 c: 1589]	1328.86 [r: 5575 c: 9339]
0.98	25.64 [r: 185 c: 307]	1018.65 [r: 4588 c: 8347]	19182.1 [r: 58943 c: 100146]

Figure 13: Performance results for the Two Agent CoverAdvance problem.

In this problem, the probability heuristic results in a significant improvement in performance. Additionally, the planner was able to find all of the possible solutions to the problem. It should be noted that this does not suggest that this planner should be used for multi-agent planning, as the planner does not attempt to merge the goals of the individual agents, nor does it distribute tasks. However, it does serve as an example of how conformant probabilistic planning can be used to solve higher level problems with multiple sub-goals.

6.1.3 Problem 3: Attack

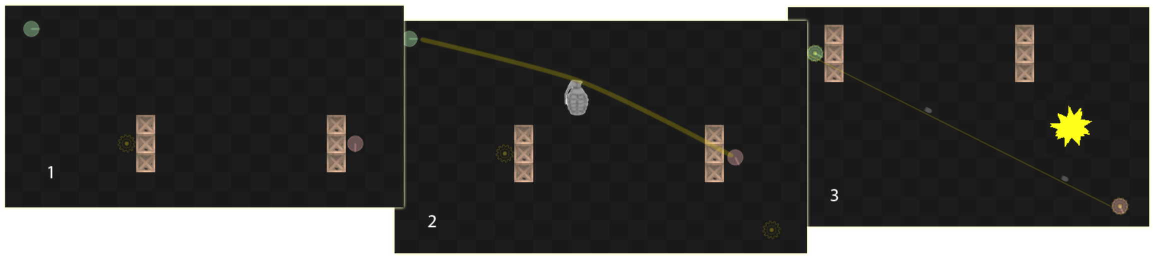

Figure 14: Attack demonstration sequence
The agent is more successful if it takes cover and flushes the enemy from cover using a grenade.

The *Attack* problem is similar to the planning example in Section 5, except with an additional action. The **Rush** action is a more expensive alternative to **Attack**, but is less likely to succeed than **AttackFromCover**. The additional strategy makes for a larger search at higher probability thresholds, as the planner must eliminate a larger number of complete, but insufficient plans. In this test, the **¬InCover** precondition of the attacking actions can be supported more than once and raise the maximum plan size to 6.

P = 0.15	p = 0.43	P = 0.48	P = 0.54	p = 0.58	P = 0.78	P = 0.83
Attack	ThrowGrenade	ThrowGrenade	ThrowGrenade	ThrowGrenade	ThrowGrenade	ThrowGrenade
	Attack	ThrowGrenade	Rush	ThrowGrenade	TakeCover	ThrowGrenade
		Attack		Rush	AttackFromCover	TakeCover
						AttackFromCover

Figure 15: Plans generated for the Attack problem

The complexity of the Attack problem lies in the number of primary sequences possible in this problem. The agent may attack, rush or attack from cover. Each of these sequences is found based on the cost and probability (Figure 15). At higher probabilities, the planner must eliminate all cheaper solutions before arriving at a viable solution. Once again, for the Buridan planner, this proves to be a difficult task. In order to reach potential solutions containing the action AttackFromCover, the planner must provide an exhaustive search of cheaper plans containing the primary action Attack. Without additional heuristics, exhausting possibilities with the primary action Attack is extremely costly. As in the previous examples, limiting the search space by only supporting designer marked preconditions results in a significant improvement in performance. While the search space for the standard Buridan algorithm explodes after searching for plans with Attack as the primary action.

Attack			
	Marked preconditions, Max actions, Prob. Heuristic	Marked preconditions, Max actions, POP first	Std. Buridan
probability	**time (ms) [r:(plans refined) c:(children generated)]**		
0.15	0.32 [r: 9 c: 14]	0.37 [r: 9 c: 14]	0.27 [r: 6 c: 11]
0.43	0.57 [r: 12 c: 17]	0.63 [r: 14 c: 19]	skips
0.48	1.12 [r: 15 c: 20]	2.17 [r: 42 c: 47]	skips
0.54	3.07 [r: 47 c: 51]	3.02 [r: 55 c: 59]	1.34 [r: 20 c: 41]
0.58	2.87 [r: 50 c: 54]	4.28 [r: 76 c: 81]	3.03 [r: 33 c: 79]
0.78	6.19 [r: 93 c: 95]	6.22 [r: 94 c: 96]	4730.12 [r: 5656 c: 16046]
0.83	8.18 [r: 98 c: 100]	10.87 [r: 129 c: 134]	4790.24 [r: 5740 c: 16383]

Figure 16: Performance results for the Attack problem.

First, it is important to note that the standard Buridan algorithm skips two of the potential solution plans. Since all of the actions in a standard Buridan plan have the same cost, the planner is not able to distinguish between two plans with the same number of actions, but a different total cost. In this case, it happens to find a solution with a higher probability, resulting in a loss of plan variety.

Additionally, the standard Buridan algorithm once again reaches a critical mass of plans in the open list. The sudden jump in planning times occurs when the planner must exhaust all of the plans involving the **Attack** and **Rush** actions (Figure 16). Without the limitation of designer marked preconditions, many child plans are generated each time a plan reaches the Buridan stage. The incomplete plans containing the actions **AttackFromCover, ThrowGrenade** and **TakeCover**, which are necessary to reach the probability threshold of 0.78 and greater, get buried at the bottom of a large open list because **AttackFromCover** has a larger number of preconditions than **Attack** and **Rush**. In general, it is very easy to create examples that force the standard Buridan algorithm to do more work than should be necessary.

6.1.4 Problem 4: Cover Exits

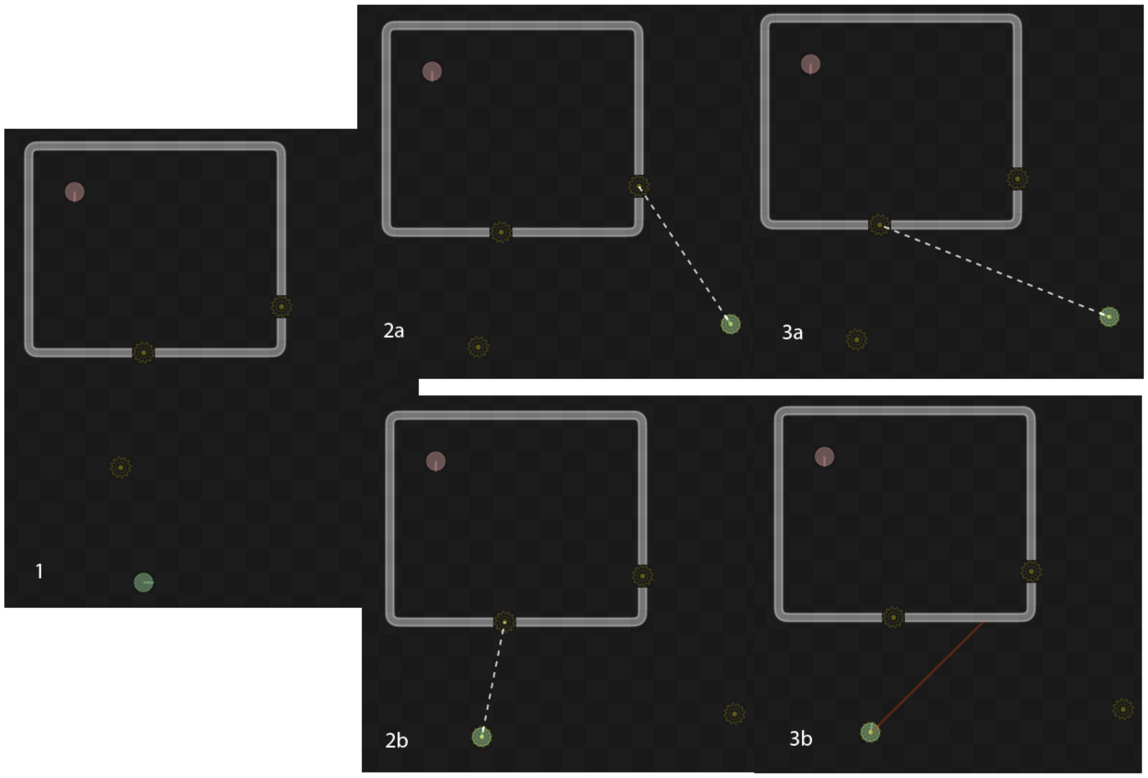

Figure 17: CoverExits demonstration sequence.
A simplified version of the CoverExits problem. The agent can cover both of the exits of the room if it makes a more expensive move. In the example discussed below there are three possible exits and three positions.

The *Cover Exits* problem presents the agent with a situation where an enemy is in a room with multiple exits. A simplified example is shown in Figure 17, although the problem evaluated is more complicated. The agent can either make moves of varying length to 3 different guard locations. From the first, the agent can only guard the closest exit. From the next location, the agent can only guard the closest and middle exits. From the far location, the agent can guard the middle and far exits. The agent in the room is least likely to exit from the closest door and most likely to exit from the far door. This problem illustrates the planner's ability to handle multiple possible world state. At the time of planning, the planner does not know which of the exits the enemy will use, so plans with high probabilities of success must take all three possibilities into account (Figure 18).

p = 0.15	p = 0.25	p = 0.4	p = 0.6	p = 0.85
Move1(agent, p2)	Move2(agent, p3)	Move2(agent, p3)	Move3(agent, p4)	Move3(agent, p4)
Guard(near)	Guard(middle)	Guard(near)	Guard(far)	Guard(medium)
		Guard(medium)		Guard(far)

Figure 18: Plans generated from the CoverExits problem.

While the performance of this problem is technically worse when using the modified planners (Figure 19), it illustrates why the planner requires that actions need to have individual costs. When planning with the standard Buridan algorithm, all actions have a cost of one. This means that the planner considers the actions Move1, Move2 and Move3 to be equivalent. In order to plan for a variety of behaviors, the planner must be able find cheaper, less successful plans. While the standard Buridan algorithm finds the highest probability solution more quickly, its inability to find all possible solutions makes it insufficient for these purposes.

Cover Exits			
	Marked preconditions, Max actions, Prob. Heuristic	Marked preconditions, Max actions, POP first	Std. Buridan
probability	**time (ms) [r:(plans refined) c:(children generated)]**		
0.15	0.67 [r: 10 c: 13]	0.67 [r: 10 c: 13]	skips
0.25	2.92 [r: 41 c: 43]	2.11 [r: 25 c: 31]	skips
0.4	4.28 [r: 50 c: 57]	3.75 [r: 37 c: 48]	skips
0.6	8.24 [r: 107 c: 107]	6.24 [r: 67 c: 77]	0.66 [r: 10 c: 13]
0.85	9.55 [r: 114 c: 119]	7.72 [r: 77 c: 92]	4.07 [r: 44 c: 55]

Figure 19: Performance results for the CoverExits problem.

Another important consideration when examining this problem is the way in which considering multiple world states and planning for a variety of agent difficulty work in tandem. The planner is able to effectively and purposefully create plans with flaws that would allow the enemy in the room to escape more easily. Conversely, high probability plans would make it very difficult for the enemy to successfully exit the room, assuming that the planner was given probabilities that accurately represent the situation. Here, the planner uses a powerful feature of probabilistic planning, representing multiple possible world states, and successfully applies it to the goals of planning for variety.

7 Implementation Details

While Buridan has been implemented in LISP, re-creating the algorithm in C++ presented some challenges. The main question is that of performance. Implemented in 1994, the runtime results presented are outdated and also likely contain some overhead due to the LISP runtime. This implementation of the Buridan algorithm, which was used to generate the plans above, is written in C++ and takes a Lua script file as input. Using a script file for the plan representation allows the user to run the algorithm on different plans without re-compiling. This retains the benefits of the Lisp implementation and adds the efficiency and compatibility of C++. The action representation used here is similar to that used in the Buridan Lisp implementation with some format changes to work with Lua. When the planner loads, the available actions, as well as the initial and goal conditions are loaded into data structures. The planner then runs and returns a partial ordered plan.

The world representation in this implementation is also defined in the Lua file. Unlike a real game implementation, the information used in the planning process is manually added. Thus, the information available to the planner is exactly what is needed, no more, no less. In a real example, the planner would likely receive some amount of useless information. If, at first glance, this excess information seems applicable to the current plan, the planner will generate additional plans, slowing the process. Therefore, the information sent to the planner would ideally be pruned to what is relevant as much as possible. On the other hand, if a single piece of information necessary to generate the expected plan is missing, the planner will not produce the desired results. Consequently, the other modules of the AI systems would have to be adjusted so that the initial state passed to the planner is complete, but as slim as possible.

Like GOAP, this implementation has C++ classes for each action type, which can then be executed until they succeed or fail. Once a plan is generated, the actions are converted into tasks and added to a task manager, similar to the Behavior Tree technique described by (Champandard, 2008). This allows the game application to create a sequence of tasks for the agent to execute using basic types of tree nodes. Chamandard describes Sequences, Selectors and Parallels as the main types of nodes. These types are borrowed here, and some new types are added. Each of these composite node types contains a set of child tasks, which themselves could be a composite task. Sequences execute their child tasks in order, and only move to the next task when the previous task completes. Selectors choose one of their child tasks to execute. Buridan does not produce plans that have a corresponding structure to Selectors. However, if there are several different variations of the same action which could all be represented with the same preconditions and effects, a selector could be used to determine which one is used in a particular instance. For example, if the planner produces a plan with the action **CoverAgent**, a selector could be used to choose a random sound clip for the agent to use. Parallel tasks have a much more direct relation to plans produced by Buridan. When two actions in a plan have the same topographical index, a parallel task is created so that both are executed at the same time. A new composite task, Observer is also added. This task executes one task until another task finishes. This is useful in situations like in the CoverAdvance problem described above. There, an Observer task is created so that one agent covers another until the moving agent reaches its goal. The **Move** task also has special behavior and finds a path using A* and then moves the agent along the path until the destination is reached. It then returns a success code to the task manager, signaling that it should move on to the next task.

The result is a framework which is easy to extend and debug. Plans generated with the planner can be relatively high level, determining the broad strokes of an agent's behavior. Once a plan is converted into a behavior tree, small scale details can be added. This could include defining non-functional behavior like selecting which character animation to play, or more functional like choosing which sub actions to use to accomplish a larger goal. While different sub actions may affect the likelihood of success of a plan, the designers of the game could tune sub actions and decide how much sub actions should differ from one another. In this way, probabilistic planning with a goal of variety allows for a substantial increase in flexibility and problem solving capability in games.

8 Future Work

Modern planners achieve levels of performance orders of magnitude better than Buridan by reducing the size of the search space. While the goal of planning for variety precludes the use of many techniques, and often forces the planner to perform a larger search than would be necessary when looking for the best plan, using an informative heuristic to rank incomplete plans would help reduce the search size as much as possible. PVHPOP, as mentioned above, utilizes the most promising of these heuristics. Since PVHPOP is a causal-link style planner, its relaxed planning graph heuristic could be directly applied to Buridan. Planning for variety means that the planner presented here must search through cheap, low probability plans before examining more expensive ones. However, the techniques used in PVHPOP could potentially allow the planner to avoid doing so. The probability estimate found by evaluating partial plans with the relaxed planning graph is an upper bound, because threats and negative effects are ignored. If the upper bound was found to be lower than the success threshold, the planner could safely skip further refinement of the plan in question. The result would be a much more direct search path to the cheapest, yet sufficiently successful plan.

There are also other options for plan assessment. Weld *et. all* (1994) only present methods for finding exact solutions. The Monte Carlo technique could prove to be much more efficient, especially on large plans and plans with actions which have many consequences. Applied to the Buridan planner, total orderings could be non-deterministically chosen rather than exhaustively calculated. Additionally, each sample would only require one world state to be maintained, rather than every state possible. To improve accuracy for a given number of samples, one could also employ particle filtering (Russel, 2003). Particle filtering is an approximation technique that attempts to limit the samples taken to the ones that are actually important. The algorithm works by assigning weights to each possible sample generated. Samples with higher weights are then more likely to be chosen for the next pass. In this way, the algorithm focuses on the most relevant data.

Despite the unfortunate complexity of the assessment process, the primary obstacle of probabilistic planning is size of the search space. One of the most promising methods of improvement mentioned by Weld is the use of information found while assessing a plan in the refinement process. During Forward assessment, for example, the planner calculates the lower bound on the probability of success. However, while doing so, it may find a plan with a high probability in one particular ordering, but that is low in all others. The lower bound alone will not represent the potential of this plan. By notifying the refinement step of the total ordering with a high success probability, the refinement step could choose to promote/demote specific actions.

9 Conclusion

Planning has been proven to be an effective way to improve agent behavior while reducing the burden on game designers (Orkin, 2006). Probabilistic planning both widens the range of solvable problems and is able to produce a variety of possible solutions. Additionally, the planner presented here retains the loose coupling of actions and goals found in Orkin's GOAP. In practice, this means designers can change the plans an agent will use by simply adding or removing actions or by changing the goal conditions. Additionally, this version of the Buridan planner allows the designer to tune agent behavior by simply modifying the success probability threshold. By limiting the size and type of problems sent to the planner, and by splitting the planning process across multiple game cycles, one could feasibly use the current implementation in a game environment.

The adjustments made to the Buridan algorithm in the application to gaming were also mostly successful. Since plan variety and planning efficiency are often adversarial goals, it becomes important to only examine partial plans that can likely be refined into a complete plan. However, to maximize potential plan variety, the planner must traverse low probability plans first. By defining which preconditions should be reinforced, using a maximum plan depth, and including probability in the heuristic, the number of plans refinements and the time spent on assessment has successfully been reduced. However, by using an informative plan ranking heuristic, further optimizing the way the information representing a partial plan is stored, and by using an approximated assessment technique such as Monte Carlo, performance could be improved even further.

Finally, by example of Champandard’s task based AI system, it is clear that the planner is able to be easily adapted to work with other modules of a game engine. Partial order plans fit very nicely into this paradigm and are able to take advantage of parallel, observer, and sequence composite tasks. Consequently, the planner presented here builds on, and sends results to proven game friendly technology, resulting in a module that could provide very interesting video game agent behavior.

References

- Blum, A., & Furst, M. (1997). Fast Planning Through Planning Graph Analysis. *Artificial Intelligence* .

- Blum, A., & Langford, J. (1999). Probabilistic Planning in the Graphplan Framework. *5th European Conference on Planning (ECP'99).*

- Blythe, J. (1999). An Overview of Planning Under Uncertainty. *AI Magazine* , 37-54.

- Brafman, R., & Hoffmann, J. (2006). Conformant Planning via Heuristic Forward Search: A New Approach. *Artificial Intelligence* .

- Champandard, A. (2008). Getting Started with Decision Making and Control Systems. *AI Wisdom*

- Champandard, A., Straatman, R., & Verweij, T. (2010). On the AI Strategy for KILLZONE 2's Multiplayer Bots. *AIGameDev.com* .

- Chan, H., Fern, A., Ray, S., Wilson, N., & Ventura, C. (2007). Extending Online Planning for Resource Production in Real-Time Strategy Games with Search. *International Conference on Automated Planning and Scheduling.* Providence, Rhode Island.

- Domshlak, C., & Hoffmann, J. (2006). Fast Probabilistic Planning Through Weighted Model Counting. *Sixteenth International Conference on Automated Planning and Scheduling.* Menlow Park, CA: AAAI Press.

- Hoffman, J., & Brafman, R. (2005). Contingent planning via heuristic forward search with implicit belief states. *15th Int. Conf. on Automated Planning and Scheduling (ICAPS)* .

- Hoffmann, J., & Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* .

- Kelly, J., Botea, A., & Koenig, S. (2007). Planning with Task Networks in Video Games. *ICAPS07 Workshop on Planning in Games.*

- Little, L., & Thiebaux, S. (2007). Probabilistic Planning vs Replanning. *Workshop on Planning Competitions: Past, Present and Future.*

- McAllestor, D., & Rosenblatt, D. (1991). Systematic Nonlinear Planning. *AI Memos (MIT)* .

- Nilson, N. (1998). STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* .

- Nilufer, O., Welan, G., & Li, L. (2006). Engineering a Conformant Probabilistic Planner. *Journal of Artificial Intelligence Research 25* .
- Orkin, J. (2006). Three States and a Plan: The A.I. of F.E.A.R. *Game Developer Conference.*
- Palacios, H., & Geffner, H. (2009). Compiling Uncertainty Away in Conformant Planning Problems with Bounded Width. *Journal of Artificial Intelligence Research* .
- Penberthy, J. S., & Weld, D. (1992). UCPOP: a sound, complete, partial order planner for ADL. *Proceedings Third International Conference on Principles of Knowledge Representation and Reasoning (KR-92).* Cambridge, MA.
- Ramakrishnan, S. P., Pollack, M., & Smith, D. (2004). Plan-graph Based Heuristics for Conformant Probabilistic Planning. *NASA Technical Documents* .
- Russel, S. N. (2003). *Artificial Intelligence: A Modern Approach, Second Edition.* Prentice Hall.
- Smith, D., & Weld, D. (1998). Conformant Graphplan. *15th National Conference of AI* .
- Sungwook, Y., Fern, A., & Givan, R. (2007). FF-Replan: A Baseline for Probabilistic Planning. *17th International Conference on Automated Planning and Scheduling.*
- Weld, D. (2010). Systematic Nonlinear Planning. *AI Magazine* .
- Weld, D., Hanks, S., & Kushmerick, N. (1994). An Algorithm for Pobabilistic Planning.