General-Purpose Planning Algorithms In Partially-Observable Stochastic Games

BY

Bryan McKenney

SENIOR HONORS THESIS

Submitted to the University of New Hampshire
in Partial Fulfillment of
the Requirements for the Degree of

Bachelor of Science

in

Computer Science

May, 2023

ACKNOWLEDGMENTS

Many thanks to:

- **Professor Wheeler Ruml**, for being a great mentor.
- **The UNH Hamel Center for Undergraduate Research**, for research and AAAI presentation funding.
- **The UNH CS Department**, for poster and AAAI presentation funding.
- **AAAI**, for presentation funding.
- **Dream Sloth Games**, and especially **Suraako**, for resurrecting *Duelyst* and letting me work on it.
- **Chuck Olson**, creator of the *Duelyst* starter AI, for his invaluable AI vs AI experimentation framework.

TABLE OF CONTENTS

ABSTRACT

General-Purpose Planning Algorithms

In Partially-Observable Stochastic Games

by

Bryan McKenney

University of New Hampshire, May, 2023

Partially observable stochastic games (POSGs) are difficult domains to plan in because they feature multiple agents with potentially opposing goals, parts of the world are hidden from the agents, and some actions have random outcomes. It is infeasible to solve a large POSG optimally. While it may be tempting to design a specialized algorithm for finding suboptimal solutions to a particular POSG, general-purpose planning algorithms can work just as well, but with less complexity and domain knowledge required. I explore this idea in two different POSGs: Navy Defense and Duelyst.

In Navy Defense, I show that a specialized algorithm framework, goal-driven autonomy, which requires a complex subsystem separate from the planner for explicitly reasoning about goals, is unnecessary, as simple general planners such as hindsight optimization exhibit implicit goal reasoning and have strong performance.

In Duelyst, I show that a specialized expert-rule-based AI can be consistently beaten by a simple general planner using only a small amount of domain knowledge. I also introduce a modification to Monte Carlo tree search that increases performance when rollouts are slow and there are time constraints on planning.

CHAPTER 0

Introduction

A *game* is a domain in which two or more players with potentially different objectives take actions to interact with a shared world. Games have been important to human cultures for thousands of years, and even our daily interactions (and those of nonhuman animals) fit the definition of a game, so it should come as no surprise that there are many different overlapping categories that games can fall into. *Cooperative games* have the players working together towards a common goal, while *strategic games* pit the players against each other by giving them opposing objectives. A special kind of strategic game is a *zero-sum game*, in which there are exactly two players and one player's loss is the other's gain. Chess and Go are examples of zero-sum games. In *real-time games*, players take their actions simultaneously, while in *turn-based games*, only one player can take actions at a time and that player switches during the course of the game. (Although I will use the terms "turn" and "ply" interchangeably in this work, a *ply* is the technical term for one turn of one player.) A *perfect-information game*, or *fully-observable game*, is one where all aspects of the game state are visible to all players at all times. An *imperfect-information game*, or *partially-observable game*, is just the opposite — not all aspects of the game state are visible to all players at all times. Poker is an example of a partially-observable game because players' hands are secret. An *open-world game*, which must necessarily be a partially-observable game, is one in which players do not even know if some parts of the game state exist or not. (When "open world" is used to describe a video game, it is usually meant that the game has a large map where the player can wander around freely, but these games also fit the aforementioned definition of open-world if the player does not know everything about the game, which is often the case.) A game is called *one-sided* if only one player has

imperfect information or does not know what exists in the world. A final distinction to make is between *deterministic* and *stochastic* games. In a deterministic game, there is only one possible outcome for a given action, but in a stochastic game there can be multiple, each with its own probability of occurring. This work focuses on the strategic turn-based partially-observable stochastic games (POSGs) Navy Defense and Duelyst.

Navy Defense is a strategic turn-based one-sided open-world POSG. In the game, a Navy ship player must defend cargo ships from submarine adversaries on a grid-board. There is no way to win for the Navy ship — it is rather a *goal of maintenance* that the cargo ships have to be protected forever — but the submarines achieve their ends by destroying all cargo ships. The submarines are invisible, which is what makes the game partially-observable, there are an unknown number of subs, which is the open-world aspect, and the subs tie-break between best moves at random, which is the stochastic element. Chapter 1 explores Navy Defense in detail.

Duelyst is a strategic turn-based zero-sum POSG. It is a card game that does not use a standard deck of cards, and instead lets players play minion, spell, and artifact cards onto a 9x5 grid board to try to destroy their opponent's General. It is zero-sum because there are two players and only one of them can win (or there can be a tie), partially-observable because each player cannot see the cards in their opponent's deck or hand, and stochastic because drawing cards is random and some cards have abilities that involve randomness. Chapter 2 explores Duelyst in detail.

In order to promote understanding of how to create artificial intelligence to play these games, I will first explain planning and search, then methods of solving strategic turn-based perfect-information games, then delve into the more complex methods of solving strategic turn-based POSGs.

0.1 Planning and Search

A *planning domain* is a problem in which an *agent* — the artificial intelligence — is trying to achieve some goal and must reason about how to do so. The agent exists as part of the

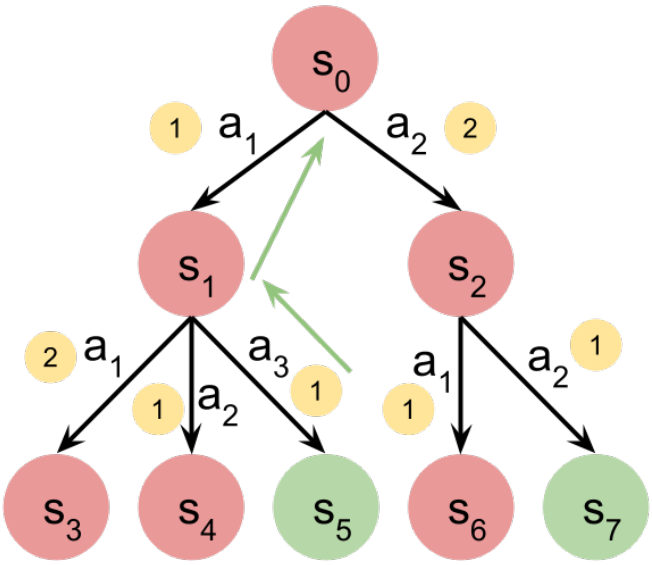

Figure 0-1: A search tree. The green nodes contain goal states.

state of the world and can change the state by taking *actions*. The set $\mathcal{S}$ of all possible states is called the *state space* and the set $\mathcal{A}$ of all possible actions is called the *action space*. A subset of the action space can be available to perform in a given state, and taking an action yields a *cost* or *reward* (i.e. negative cost) to the agent as well as (potentially) modifying the state. In partially-observable domains, the agent cannot see the state it is in, but instead gets an *observation* about that state which leaves out some information. The set $\mathcal{O}$ containing all possible observations is called the *observation space*. A *plan* π is a sequence of actions that leads to a state where the agent's goal conditions are satisfied. An *optimal plan* π^* is one that finds a goal while incurring minimal cost/gaining maximum reward. If the cost for every action is 1, the optimal plan will be one that finds a goal in the fewest actions possible.

A common way of representing a planning domain is as a *Markov decision process* (MDP). In a MDP, given a state and an action the agent can take, the next state (or probability distribution over next states if the domain is stochastic) can be found without needing to know what other states came before. This is called the *Markov property*. A *partially-*

observable Markov decision process (POMDP) is a MDP with imperfect information, so each state is associated with an observation.

Planning algorithms (also called *planners*) are often general-purpose because they use *search* algorithms to explore the state space regardless of the content of the states or the nature of the actions. They simply search until a specific goal is found and follow the general approach of minimizing cost/maximizing reward to select the optimal plan. A *heuristic function* can be utilized to estimate the cost to the goal from a given state to help speed up the search, and heuristics usually require domain knowledge, but they are separate from the planner itself and can easily be swapped out when planning in a different domain. Figure 0-1 gives an example of a *search tree*. The red circles are nodes containing non-goal states, the green circles are nodes containing goal states, the black arrows are actions, and the yellow circles are action costs. s_0 is the initial state, and both s_5 and s_7 are goal states, but reaching s_5 incurs a total cost of 2 and reaching s_7 incurs a total cost of 3, so the plan to s_5 is optimal. Backtracking from this goal (shown by the green arrows) to the initial state, one can reconstruct the plan, which is to take action 1 and then action 3.

Computing an optimal plan and then executing it is called *offline planning*. When planning in a MDP with its stochastic state transitions or a game, offline planning can still be applied by generating an optimal *policy* of what action to take in any state that the agent could end up in, but if the state space is too large, this approach becomes intractable. In a POMDP or POSG, the agent does not know what state it is in, so an optimal policy would have to be based on the agent's belief about the current state, but the same problem exists when the state space is large — a policy would be too large to compute. In these situations, *online planning* is the solution. An online agent takes one action at a time, making observations in-between and using the information that it has to decide its next move. Online algorithms are usually *suboptimal*, but since solving very difficult problems optimally is impossible, "suboptimal but good" is good enough.

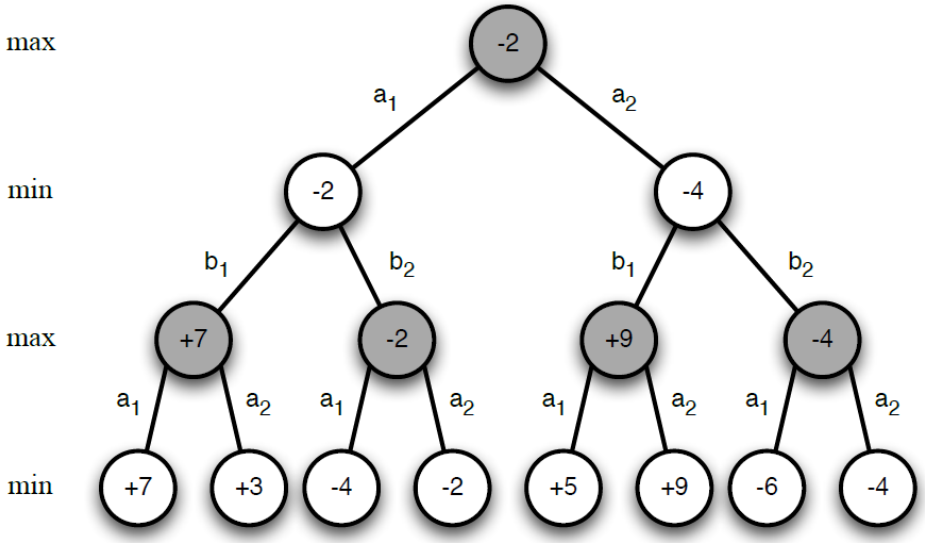

Figure 0-2: A minimax game tree. Image from Gelly et al. (2012).

0.2 Solving Perfect-Information Games

Games, even when they have perfect information, are not MDPs because they involve multiple agents, so from the perspective of one agent, the state transitions do not always depend on its action and the current state. Each ply in a game may be a MDP, assuming the Markov property is satisfied, but the game as a whole is not. Games also may not have action costs/rewards, but only a reward at terminal states (when the game is over) for winning, losing, or tying. Most zero-sum games, for instance, have a reward of 1 for a win, 0 for a tie, and -1 for a loss (chess is an exception, as it traditionally has a reward of 1 for a win, 0.5 for a tie, and 0 for a loss, but this is essentially the same thing). For these reasons, most normal search techniques do not apply to solving games, but there are others that do. I will discuss two of these approaches: minimax and Monte Carlo tree search.

0.2.1 Minimax

Minimax is an algorithm designed for two-player strategic turn-based perfect-information games (Gelly et al., 2012). It involves optimally planning moves based on the assumption

that the opponent will also play optimally (if they do not in reality, so much the better). The planning player wants to win the game, which means they want to reach an end-game state with maximum reward (1 in a zero-sum game). Therefore, they are represented in a minimax *game tree* as the *max* player. Since the opponent has an opposing goal to the max player, they are represented as the *min* player, who wants to reach an end-game state with minimum reward. This is because the tree is from the max player's perspective, so the minimum reward for the max player is the maximum reward for the min player. An example of a minimax game tree is shown in Figure 0-2. The tree consists of nodes (the circles) which each represent a state and display its calculated value. The nodes are connected by actions that the players can take. From the initial state (root node), the entire game tree is built and then the values from the end-game states (the leaf nodes) are backpropagated up the tree to the root (taking either the min or max of the children's values at each parent node), which then tells the player which action they should take. In this example, action 1 is chosen because the root player is the max player and $-2 > -4$. The best end-turn state ($+9$ reward) is only reachable if the player takes action 2, but the opponent would then have to take action 1, which they would not do if they were playing optimally.

When a game is stochastic, it can still be solved using minimax by introducing the *chance* player. Since chance is not aligned with either player, the values of a chance node's children are averaged when constructing the tree. This yields the expected value of the random action's outcome. For this reason, minimax with chance nodes is called *expectiminimax*.

Since minimax builds the entire game tree and discovers the best action to take in any situation, it is an optimal offline algorithm. When a game's *branching factor*, or number of actions available in each state, is too large or it takes too many actions to reach the end of the game, it is infeasible to construct the whole tree. A solution is to make minimax an online suboptimal algorithm by cutting off search at a certain tree depth and applying a *static evaluator* at leaf nodes. A static evaluator is similar to a heuristic, although it does not estimate the cost-to-go but rather estimates the value of a state based on certain features of that state (therefore, it is domain-specific). Figure 0-2 can also be looked at as

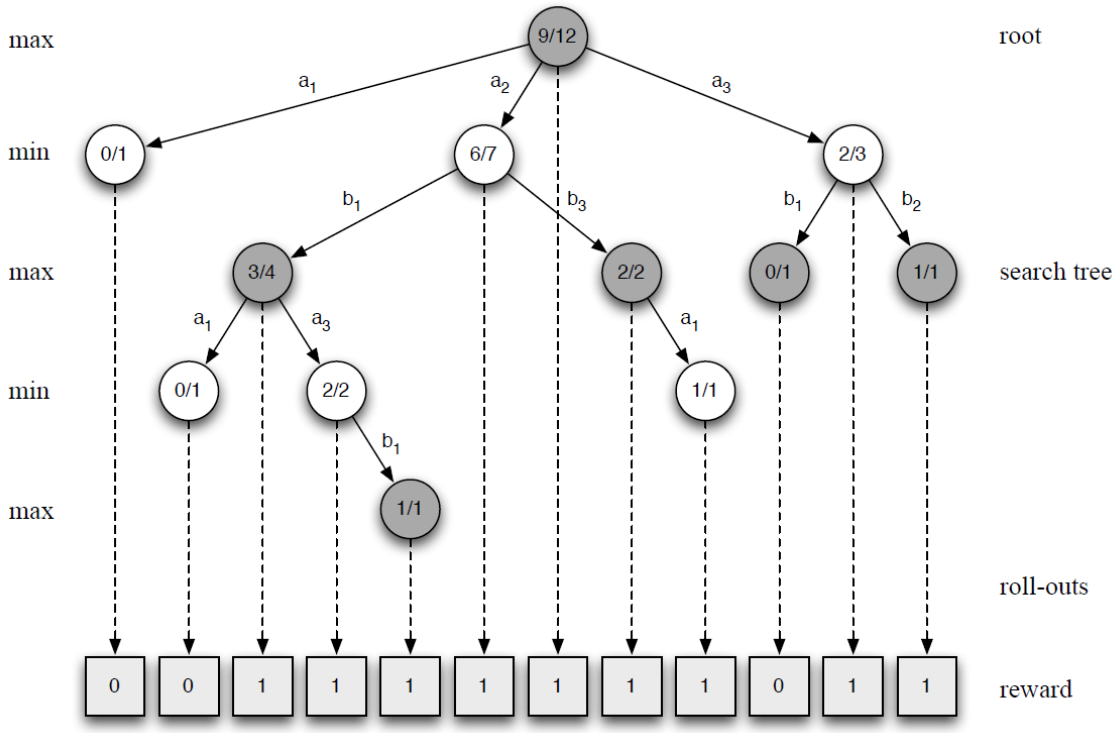

Figure 0-3: A MCTS game tree. Image from Gelly et al. (2012).

an online minimax tree with search depth 3. It could even be for a zero-sum game, since static evaluations do not represent rewards.

0.2.2 Monte Carlo Tree Search

Monte Carlo tree search (MCTS) was designed for MDPs, not games, but it has been applied to games such as Go with great success (Gelly et al., 2012). The idea behind MCTS is to learn where the good parts of the game tree are and focus on exploring those rather than the whole thing (it is a solution to the multi-armed bandit problem). It builds a search tree of alternating state and action nodes, each of which contain a visitation count N and an estimated value V (initially both set to 0). Repeated simulations are performed from the root of the search tree using the following procedure:

1. **Tree search:** Generate child action nodes unless they already exist, then choose an

action node using the *UCB* algorithm (explained below). Simulate to find the next state obtained from taking the chosen action in the current node's state. If this next state exists in the tree as a child of the chosen action node, repeat this step from that state node. Otherwise, add a new state node to the tree as a child of the chosen action node.

2. **Rollout:** From the newly added state node, perform a *rollout* (simulate taking one action at a time following a certain policy) until a terminal state is reached. The reward at this terminal state becomes the value of the new state node and its visitation count is set to 1.

3. **Backpropagate:** Starting from the parent of the newly added state node, backpropagate to the root, incrementing visitation counts and using some method to update node values (e.g. take the minimum/maximum of child values at state nodes and the mean at action nodes, since they are akin to chance nodes) along the way.

This process is continued until time is up or a certain number of simulations are run, and then the action with the highest value or visitation count is taken. The UCB (also called UCT in this context) algorithm selects which action to take during tree search; it incorporates the visitation count to add an exploration bonus (weighted by the parameter C) to nodes that have not been explored much. The score for each child node (action) is calculated by:

$$V_{child} + C \cdot \sqrt{\frac{\log N_{parent}}{N_{child}}} \quad \text{if } N_{child} > 0, \text{ else } \infty$$

The action with the highest score will be selected. The infinite score for an unexplored node ensures that all actions are tried at least once. As for rollouts, a common approach is to take random actions, although a smarter domain-specific policy can lead to better results (Gelly et al., 2012).

Figure 0-3 shows a MCTS game tree. The game could be chess, because the terminal state rewards are 1 and 0, and it is deterministic, which is why there are no action nodes — when an action only leads to a single state, all of the nodes in the tree can be states,

and UCB selects a state node instead of an action node during tree search.

MCTS is theoretically guaranteed to converge to the optimal solution if given enough time (Gelly et al., 2012).

0.3 Solving Partially-Observable Stochastic Games

Partially-observable stochastic games are not as easily solved as perfect-information games, for the agent does not know what state it is in. Let us consider a simple example of a POSG and how to solve it optimally.

Example POSG You are walking through the woods when you encounter an aggressive bear of unknown species. Your goal is to survive the encounter, and the bear's goal is to kill you (perhaps you got too close to its cubs). You are backed up against a tree, so you cannot run. Your only options are to attempt to climb the tree or to attempt to intimidate the bear. You know the following facts: (1) there are only two species of bear, the black bear and the grizzly bear — 40% of all bears are black bears and the other 60% are grizzly bears; (2) black bears can climb trees, while grizzly bears cannot; (3) grizzly bears are harder to intimidate than black bears — you have a 40% chance to succeed at intimidating a black bear and a 30% chance to succeed against a grizzly bear; (4) you are not great at climbing trees and have a 50% chance to fail. You only have time to take one action before the bear mauls you. What do you do?

Solution Figure 0-4 shows how to construct expectiminimax game trees to solve this problem. There is one tree for each possible world you could be in: one where the bear is a black bear and the other where the bear is a grizzly bear. You are the max player and the bear is the min player, and since this is a zero-sum game, the terminal state rewards are 1 or -1. You go first with the action to either climb the tree or intimidate the bear, your action has a chance to fail, and then the bear can either attack you or leave. However, if it is playing optimally and really wants to hurt you, it will always attack when it has the option. It only does not have the option to attack you if you successfully intimidate it or successfully climb the tree and it happens to be a grizzly bear, and so these cases are the only chance you

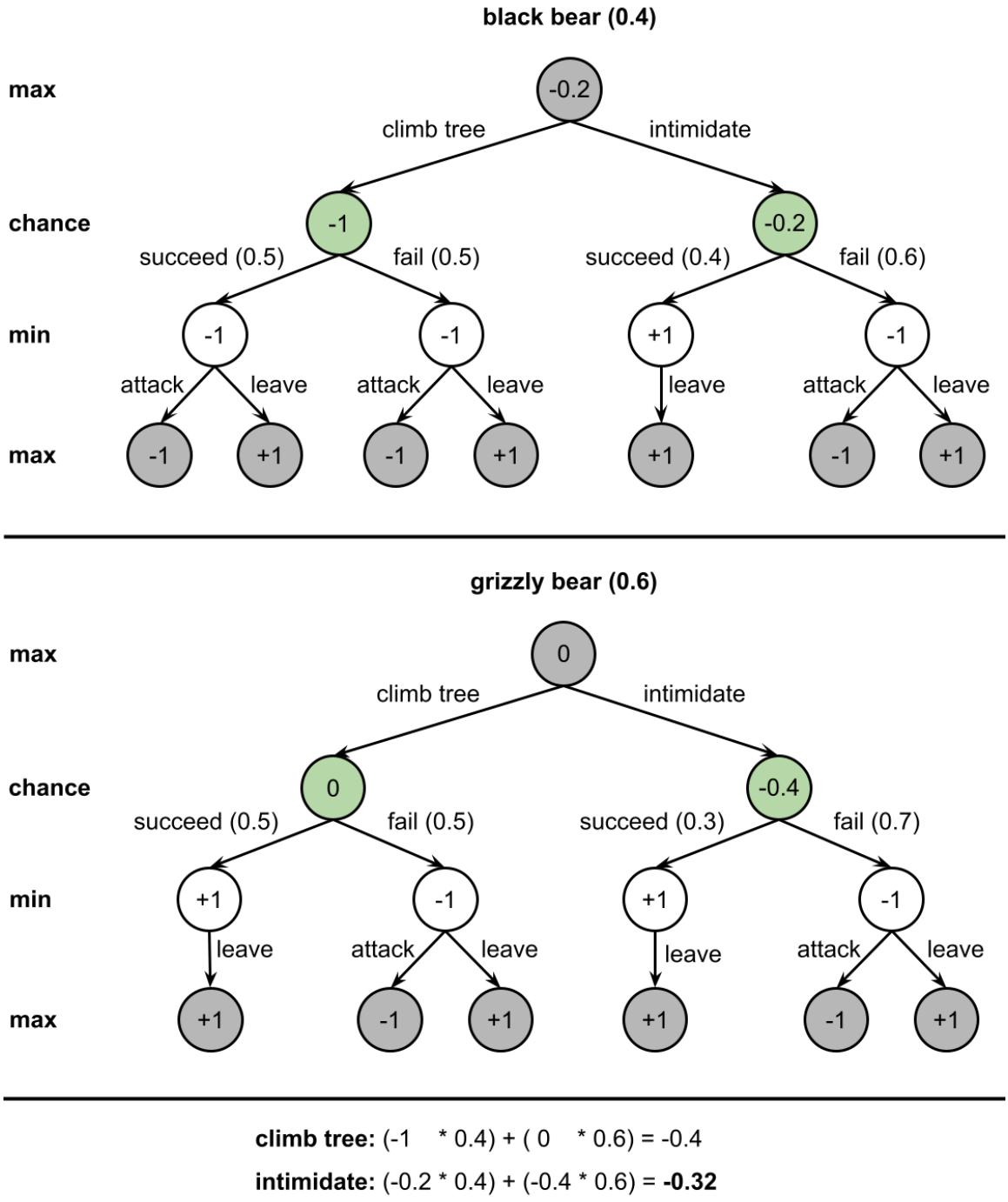

Figure 0-4: Complete game trees for calculating the optimal solution to a POSG where you are trying to survive an encounter with a bear that could either be a black bear or a grizzly bear.

have of winning. If you had a flash of insight that black bears are black and grizzlies are brown and thus were able to tell which bear you were facing, it would be easier to figure out what to do. You would only need one of these trees, and the value of the root node would tell you which action to take. If you knew you were facing a black bear you should try to intimidate, but if you knew you were facing a grizzly bear you should try to climb the tree. However, since you did not have such a flash of insight, you have to calculate the expected value of each action across both possible worlds, summing the expected value of the action in each world multiplied by the probability of the world. The calculation for each action is shown at the bottom of Figure 0-4. It turns out that you should attempt to intimidate the bear despite the fact that grizzlies are more common, because it gives you a chance of survival against either bear, while climbing a tree only saves you against a grizzly and leads to certain demise against a black bear. That said, the expected value of the intimidate action is still negative, so this may not be your day regardless of your optimal POSG tree calculations.

This simple example required two depth-3 game trees to solve optimally, but what if it were more realistic, and there were more than two species of bear, more than two actions to take, and the chance to take multiple actions before winning or losing? The number of trees, width of the trees, and depth of the trees would increase, and for a complex enough domain it would become intractable to solve. To make matters worse, what if you did not know the probability of each world or of your actions failing? This method would not work at all. Such problems are common to POSGs.

The probability distribution over bear species is an example of an exact belief over the state of the world. In this section, I will discuss general methods of modeling belief and then describe the POMCP algorithm, which can find suboptimal solutions to complex POSGs. There are many other ways to solve POSGs, some of which I shall discuss in the following chapters.

0.3.1 Belief Modeling

An agent in a partially-observable domain must have some way of estimating what the current state of the world is in order to plan intelligently. The only information it has available to it is the history of actions taken and observations gathered while it was operating in that world. This information can be compiled into an *information set* or *belief state*, which is a probability distribution over possible states.

There are several ways to model and maintain a belief state. Fox et al. (2003) review methods that use *Bayesian filtering* for POMDPs. These start with some *prior* belief and apply Bayes' Law to update probabilities when new information from an observation becomes available. An *exact* belief state tracks a probability for every state in the state space, while a *particle filter* belief state keeps a fixed-size set of possible states (called *particles*) and updates or removes them as observations are made (Fox et al., 2003). When states are found to be impossible and removed, they must be replaced by states consistent with the belief, and these can be randomized as much as possible to help the agent explore new possibilities. This is called *particle rejuvenation* (Silver and Veness, 2010). Particle belief states are faster and more flexible than other methods and can converge to the true probability distribution (exact belief) even if it is very complex (Fox et al., 2003). There are two versions of particle filters — weighted and unweighted. A *weighted* particle filter tracks the probability of each particle, while an *unweighted* one does not (a uniform distribution over particles is assumed). However, an unweighted particle filter can contain duplicate states, which approximates weights. Unweighted particle filters are useful when working with a *black box* simulator of the world, where none of the state transitions or their probabilities are known (Silver and Veness, 2010).

0.3.2 Partially Observable Monte-Carlo Planning

Partially observable Monte-Carlo planning (POMCP) is an algorithm invented by Silver and Veness (2010) to generalize MCTS to POMDPs. Along with two POMDP domains, Silver and Veness (2010) test POMCP on a POSG domain, *pocman* (partially-observable

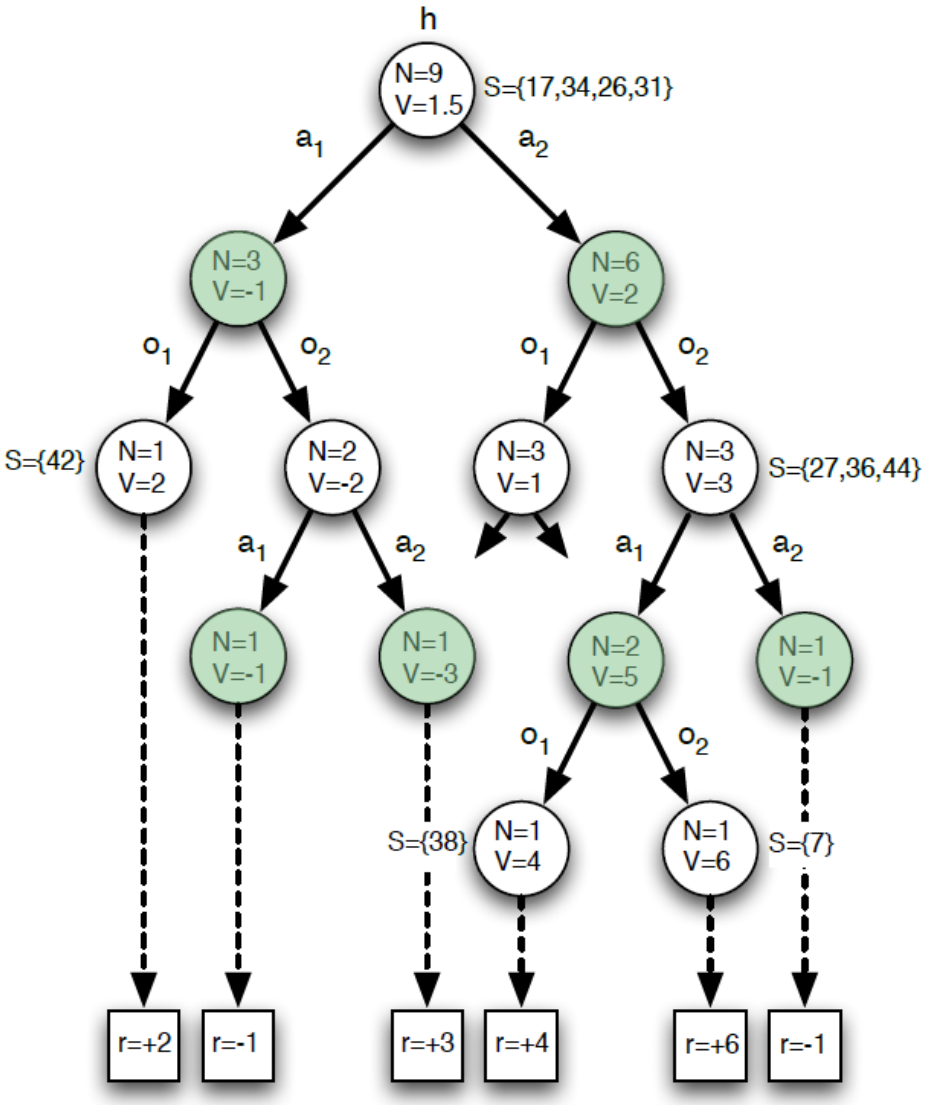

Figure 0-5: A POMCP search tree. Image from Silver and Veness (2010) (colorized).

Pac-Man) and get good results. POMCP works just like MCTS except it has observation nodes instead of state nodes and these each store an unweighted particle filter belief state S. Each simulation starts at a state sampled from the root node's belief state, and, on the way through the search tree, if a new state is discovered that matches an existing observation, that observation node's belief state is updated to include that state instead of creating a new node (only when a new observation is found is a new node created). After simulations are done, an action is chosen and executed, and an observation is obtained, if that observation is in the search tree, it becomes the new root. This method of preserving part of the tree between actions can be applied in MCTS as well. Like MCTS, POMCP is theoretically guaranteed to converge to the optimal solution if given enough time, as long as the belief state is correct.

Figure 0-5 shows a POMCP search tree. Action nodes are colored green and observation nodes are white. It should be noted that this is not necessarily a game tree — it does not show the max and min players. It could be part of a ply if the ply consists of multiple actions, but, either way, POMCP can be used for a game just as MCTS can, by taking the max or min depending on the player when backpropagating.

0.4 Summary

I introduced partially-observable stochastic games and general-purpose planning and then discussed two algorithms for solving perfect-information games and one for solving POSGs. The next two chapters will exhibit how general-purpose planning algorithms perform against specialized ones in two very different POSG domains — the Navy Defense simulation and the card game Duelyst.

CHAPTER 1

Navy Defense

Molineaux, Klenk, and Aha (2010); Klenk, Molineaux, and Aha (2013) introduce a framework for solving online open-world problems called *goal-driven autonomy* (GDA), in which the agent manages a set of goals while trying to achieve them. In GDA, goal reasoning is made explicit and separated from planning. They test their GDA-driven agent, ARTUE, on three Navy-themed POSG domains, two of which involve a Navy ship agent and a hidden submarine adversary, and they claim on the basis of these results that GDA is necessary to achieve high performance in such scenarios.

In this chapter, I strengthen and extend the argument of Paredes and Ruml (2017) that GDA is not necessary. I use a domain more similar to that of Molineaux, Klenk, and Aha, with a Navy ship agent and an unknown number of hidden submarine adversaries that are trying to destroy cargo ships, to show that a much simpler pure-planning approach called *hindsight optimization*, in which the agent samples possible worlds and plans in all of them, works just as well as GDA. Interestingly, I find that my planner gives rise to patrolling behavior without the need to explicitly program that behavior. A Navy ship agent using hindsight optimization can imagine possible actions of possible adversaries and how its actions might thwart them, thereby implicitly employing an intelligent patrolling strategy to protect the cargo ships without needing to explicitly reason about its multiple goals. I also find that the agent can implicitly switch goals in the face of unexpected events.

In the following sections, I shall review previous work on goal-driven autonomy and hindsight optimization, explain how planners can reason about goals implicitly, describe the Navy Defense domain and experimental results, and lastly consider limitations and possible extensions.

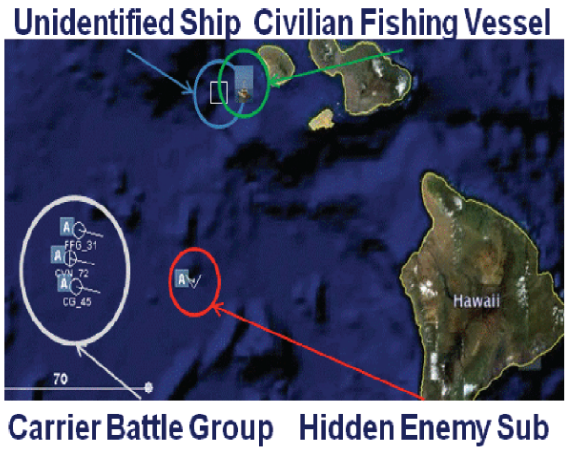

Figure 1-1: A screenshot of the Tactical Action Officer (TAO) Sandbox showing an invisible submarine adversary. Image from Molineaux, Klenk, and Aha (2010).

1.1 Background

In this section, I explain goal-driven autonomy and hindsight optimization in the context of previous work.

1.1.1 Goal-driven Autonomy

Molineaux, Klenk, and Aha (2010) observe that agents in partially-observable, open-world, adversarial, stochastic domains with continuous time and space (such as video games and simulations) need a way to deal with unexpected events that ruin plans. They introduce the goal-driven autonomy (GDA) algorithm framework as the solution to this problem. A diagram of GDA can be seen in Figure 1-2. GDA has three sub-systems: a Planner, a State Transition System, and a Controller, the last of which deals with goal reasoning. The Controller has four components: 1) The Discrepancy Detector, which compares the observed state to the expected state to find discrepancies; 2) The Explanation Generator, which hypothesizes causes for the discrepancies based on what it knows about the environment; 3) The Goal Formulator, which creates new goals based on the explanations; and 4) The Goal Manager, which prioritizes goals. Molineaux, Klenk, and Aha claim that only GDA

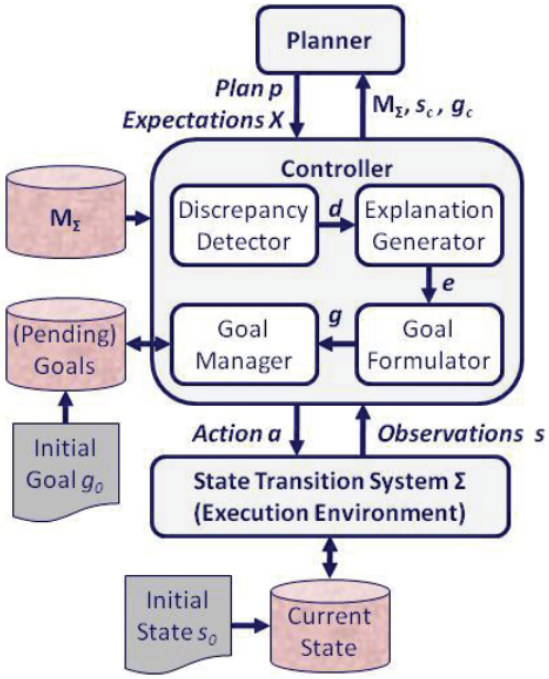

Figure 1-2: The goal-driven autonomy (GDA) framework. Image from Molineaux, Klenk, and Aha (2010).

simultaneously relaxes four classical planning assumptions — deterministic environments, static environments, discrete effects, and static goals.

They introduce a specific instantiation of GDA called *Autonomous Response to Unexpected Events* (ARTUE). For its Planner, ARTUE uses a Hierarchical Task Network (HTN) planner that predicts the future to anticipate exogenous events. ARTUE's Goal Formulator relies on domain-dependent *principles* to map explanations of discrepancies to new goals, and the Goal Manager prioritizes goals based on their hard-coded intensity levels. Molineaux, Klenk, and Aha test ARTUE with success on three scenarios in the Tactical Action Officer (TAO) Sandbox, a naval simulation in which the agent is a Navy ship. These three scenarios are: 1) *Scouting*, in which the initial goal is to identify nearby ships until an unexpected submarine attack adds the goal of identifying and destroying the sub; 2) *Iceberg*, in which the initial goal is to transport cargo between points until lightning strikes an iceberg and a severe storm arises, adding the additional goals of seeking shelter and rescuing members of a sinking ship; and 3) *SubHunt*, in which the goal is to seek and destroy a submarine while also sweeping mines that it lays. An example of the *Scouting* scenario is shown in Figure 1-1.

1.1.2 Hindsight Optimization

Hindsight optimization is a planning algorithm that finds suboptimal solutions to problems involving uncertainty by sampling possible worlds, each representing a possible resolution of the uncertainty, and planning in those. It was developed by Chong, Givan, and Chang (2000) for network control, introduced to the planning community by Yoon et al. (2008) and has since been successfully applied to a wide range of problems, from manufacturing and unmanned aerial vehicle flight patterns (Burns et al., 2012) to robot search-and-rescue and making omelettes (Kiesel et al., 2013). While hindsight optimization was not designed to be used in games, it is also known by another name, *perfect-information Monte Carlo search* (PIMC), which has been applied to games including Skat (Furtak and Buro, 2013) and also makes up an important part of Ginsberg (2011)'s GIB algorithm for Bridge. I

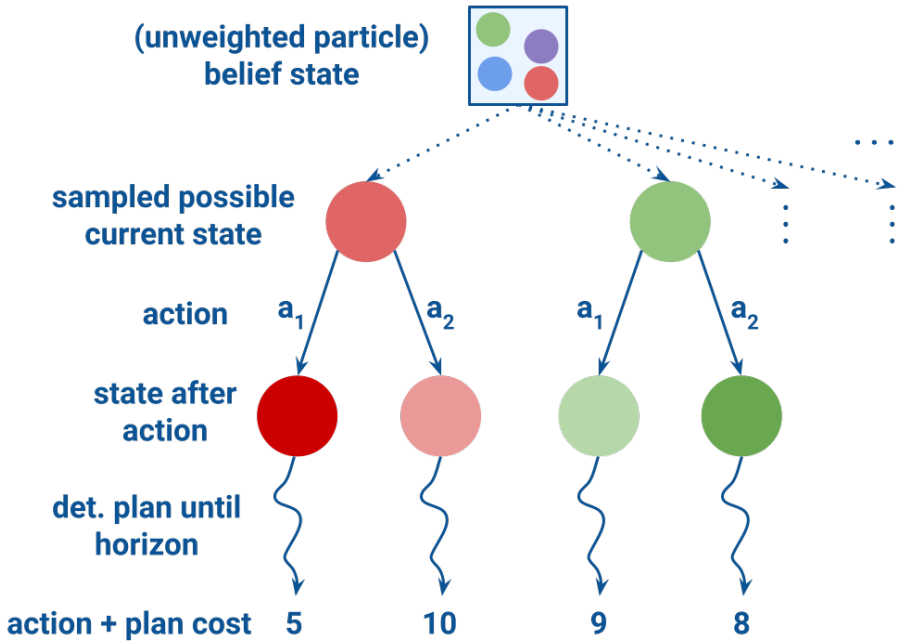

Figure 1-3: A hindsight optimization search using an unweighted particle filter belief state.

use the name hindsight optimization because, even though the Navy Defense domain is a game, it is formulated as a continuous planning problem with costs for certain events and opponents' moves are part of the transition function.

Paredes and Ruml (2017) use hindsight optimization to argue against Molineaux, Klenk, and Aha (2010)'s claim that goal reasoning needs to be separate from planning for complex domains. They create a partially-observable, open-world, online, stochastic, multi-unit, adversarial domain called Harvester World to aid their argument. In this grid-based domain, the agent controls a Harvester and a Defender, there is an Enemy that can be seen from one unit away from them but is otherwise invisible, and there are also hidden obstacles and food around the map. The agent knows the Enemy's policy. Paredes and Ruml successfully use hindsight optimization in three different Harvester World scenarios and argue that it implicitly has all of the components of goal-driven autonomy. Hindsight optimization is a type of *multilevel planner*, as the high-level deterministic planner acts as a heuristic function for the low-level hindsight algorithm, and this can be viewed as reasoning about goals and

Algorithm 1 Hindsight Optimization(o, N, H)

```
1: sampleStates ← hallucinate(N)
2: for action a applicable in o do
3:     sampleCosts ← [ ]
4:     for state s in sampleStates do
5:         s', aCost ← f(s, a)
6:         c ← aCost + planCost(s', H)
7:         append c to sampleCosts
8:     Q(o, a) ← mean(sampleCosts)
9: return argmin_a Q(o, a)
```

Algorithm 2 planCost(s, H, $t \leftarrow 0$)

```
10: if t = H then
11:     return 0
12: for actions a applicable in s do
13:     s', aCost ← f(s, a)
14:     costToGo ← planCost(s', H, t + 1)
15:     Q(s, a) ← aCost + costToGo
16: return min_a Q(s, a)
```

choosing the best one to pursue.

Like Paredes and Ruml, I use a domain (described more fully in Section 1.3) that is grid-based, partially-observable, open-world, online, stochastic, and adversarial. The adversaries can be detected if they are one space away from the agent. However, my domain has more similarities with TAO Sandbox than Harvester World does: it has an unknown number of adversaries, instead of just one, and it has only one unit controlled by the agent, instead of two. Additionally, the agent has an inaccurate (but reasonable) model of the adversaries, which makes things more challenging. I am the first to show that hindsight optimization can lead to emergent patrolling behavior, which can be seen as a strategic type of goal of maintenance.

Algorithm 1 outlines hindsight optimization. The algorithm first samples a certain number of states from the agent's belief state (line 1). A seed is associated with each state that will be used to resolve stochastic effects. Then, in each of these possible worlds, each

possible action is tried and a deterministic planner is run on the resulting states up to a certain horizon (line 6). The action that led to the lowest average cost across the sampled worlds is chosen (line 9).

Algorithm 2 outlines a basic deterministic depth-first horizon-limited planner. I use branch-and-bound to improve efficiency. (My implementation is sufficiently fast and I have not optimized runtime — child ordering using a heuristic function would likely make the planner even faster.)

Figure 1-3 shows an example of the hindsight optimization procedure. In the figure, the belief state is an unweighted particle filter, but this does not have to be the case. The green and red states are sampled from the belief state and then the two available actions are tried in both states. After deterministic planning to some horizon in each resulting state, action 1 has a total cost of 14 across both worlds (7 on average) and action 2 has a total cost of 18 (9 on average), so action 1 is chosen. This is very similar to the example of optimally solving a POSG I gave in Chapter 0, except it is not optimal. Hindsight optimization, especially when it uses a particle belief state, is like a simplified version of POMCP. However, POMCP is asymptotically optimal, while hindsight optimization has no guarantees of bounded suboptimality.

1.2 Implicit Goal-driven Autonomy

GDA has a dedicated sub-system for goal reasoning and requires a host of hard-coded goals and principles, but this is not the only way to achieve autonomous goal-driven behavior. Another way is to abstract the agent's goals down to cost values that are incurred by certain events and then make its single goal to minimize cost. When paired with an appropriate planner and a belief state that tracks knowledge about the environment (fulfilling the tasks of the Discrepancy Detector and the Explanation Generator), these costs allow trading off between multiple goals and there is no need for a Goal Formulator or Goal Manager — goal formulation and management will be done implicitly while following the prime directive to minimize cost. This eliminates the need for a separate Controller. I present hindsight

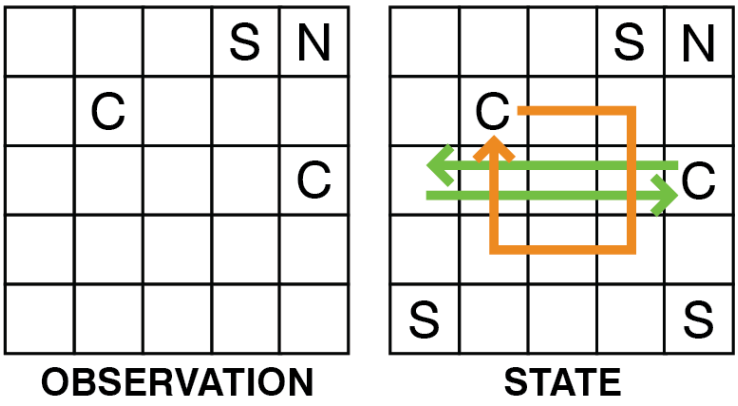

Figure 1-4: An observation and the underlying state in a Navy Defense world. One submarine is still visible in the observation because it is within the Navy ship's one-space sonar radius.

optimization as just one possible appropriate planner — POMCP or another approximate POMDP planner could be an option as well. To provide empirical evidence for my position, I ran an experiment with an agent using hindsight optimization in a simple domain, described below, which shares similarities with the TAO Sandbox domains used by Molineaux, Klenk, and Aha (2010).

1.3 The Navy Defense Domain

In this domain, a Navy ship must defend cargo ships from being destroyed by hidden submarines for as long as possible in a grid-world of variable size. There is one Navy ship (the agent) and a variable number of cargo ships (allies) and submarines (adversaries) in different starting positions. A 5x5 example is shown in Figure 1-4 with one agent/Navy ship (marked N), two cargo ships (C) to protect, and three invisible submarines that are trying to destroy them (S). The arrows show the fixed paths that the cargo ships move along. The Navy ship, cargo ships, and submarines each occupy a single space on the grid and have 2 health. When a ship or sub is reduced to 0 health, it is destroyed (removed from the world). Multiple ships and subs can occupy the same space. The agent can only observe

submarines that are within a 1-space sonar radius (including diagonals) of it, and it does not know how many submarines are in the world (but it does know the maximum number that there could be). The agent has unlimited time to contemplate its next move while the world stands still. After the agent takes an action, cargo ships and then submarines (in the order that they were created in) make their moves. All submarines decide what they will do before any of them acts. We define On Hit to mean that a vessel has sustained damage but still has more than 0 health afterwards, and On Destroy to mean that it is destroyed. The ships and subs work in the following ways:

Navy Ship (Agent) Can stay still or move one space in any of the four cardinal directions (within the grid boundary), then deals 1 damage to all submarines within sonar radius. Moving incurs a cost of 1; On Hit, incurs a cost of 10; On Destroy, incurs a cost of 40.

Cargo Ship (Ally) Travels automatically in a rectangle, staying a constant distance from the edge of the grid, in a predetermined direction (clockwise or counterclockwise). On Hit, incurs a cost of 20; On Destroy, incurs a cost of 80. This reflects the idea that the cargo ships are more valuable than the Navy ship; for example, if there is one time step left and either the cargo ship or the Navy ship will be destroyed, the agent would be expected to sacrifice itself for its mission. The future value of the Navy ship, insofar as it can protect cargo ships over time, is up to the planner to reason about.

Submarine (Adversary) Moves orthogonally, avoiding the Navy ship's sonar radius, to the nearest point of intersection along a cargo ship's route. Can also stay still to lie in wait. Instead of moving, the sub can attack the space it is on or one of the four spaces adjacent to it, dealing 1 damage to all ships there. If inside the Navy ship's sonar radius, will move out of it, if possible, or attack the Navy ship's space otherwise. Breaks ties between equally good moves randomly. The agent believes that subs choose to target a cargo ship at random and then hunt it down until it is destroyed before switching target, but this is not the true behavior; a sub will switch targets to whichever cargo ship it can intercept more quickly.

For small worlds (e.g. 5x5), the agent can keep an exact belief state (the probability of each possible current state), but for larger worlds an unweighted particle filter belief state

is ideal.

1.3.1 Navy Defense vs. TAO Sandbox

Like the TAO Sandbox, the Navy Defense domain is online, partially-observable, open-world, stochastic, adversarial, infinite horizon, and features a Navy ship agent. Navy Defense is grid-based, however, while TAO Sandbox is not.

Navy Defense is similar to the *Scouting* scenario because there is a hidden submarine that attacks ships and the agent can see it only by using sensors and is able to destroy it. In *Scouting*, however, the agent gets rewarded for destroying the submarine, which is not the case in Navy Defense, and in Navy Defense there can be more than one submarine.

Navy Defense is similar to the *SubHunt* scenario as well, but in *SubHunt* the goal is to find and destroy the submarine, while in Navy Defense destroying subs is not necessary as long as the cargo ships are protected. The mines in *SubHunt* can be considered an unknown number of static adversaries, and in Navy Defense there are an unknown number of dynamic adversaries.

1.4 Experimental Results

I conducted three experiments in the Navy Defense domain. The first was to tune hindsight optimization's parameters. The second was to discover if the particle belief state was working by comparing it to an exact belief state and to test if more particles would help. The third was to see how well hindsight optimization performed against various baseline algorithms.

1.4.1 Tuning Hindsight Optimization

Hindsight optimization with an unweighted particle filter belief state has three parameters: the number of particles to maintain, the number of states or worlds to sample in making each decision, and the horizon for the deterministic planner. I tried all combinations of 10 and 30 particles, 1, 3, 10, 30, and 100 world samples, and horizons of 1, 3, 5, 10, and 15.

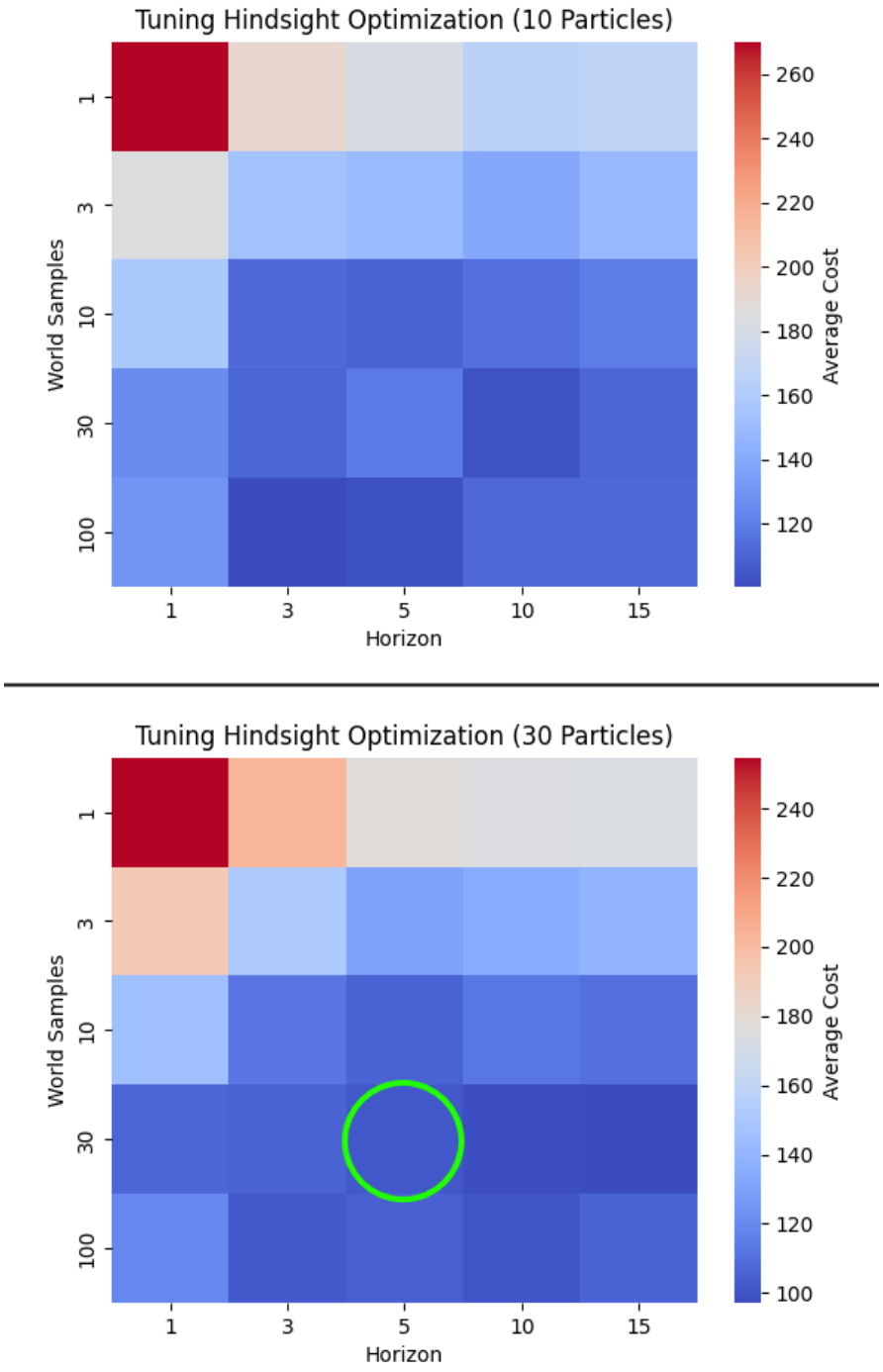

Figure 1-5: Heatmaps showing the average cost hindsight optimization incurred with different parameter values.

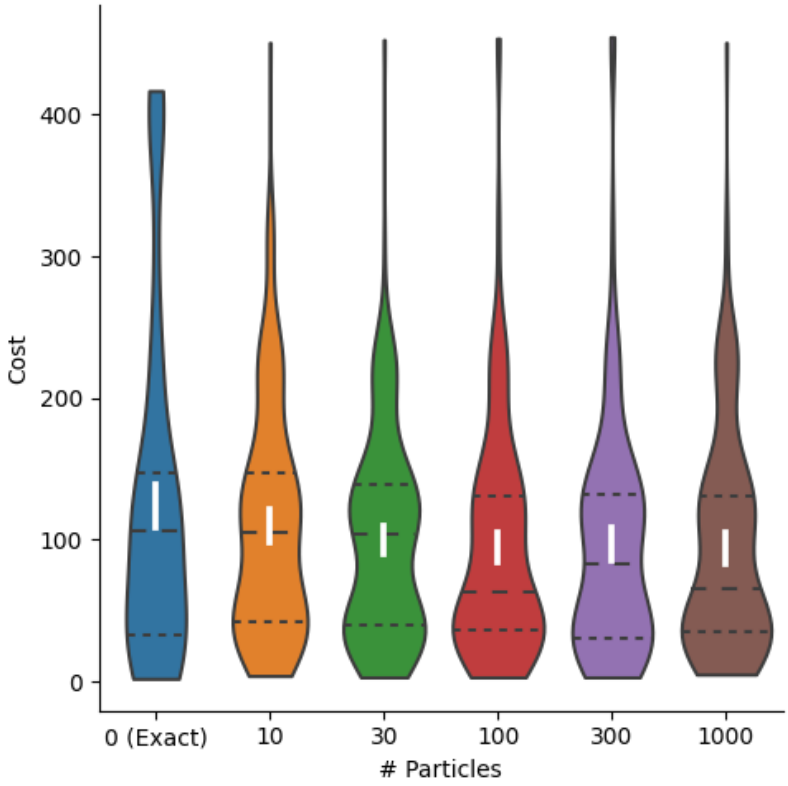

Figure 1-6: The distribution of costs incurred by hindsight optimization using an exact belief state or various numbers of particles.

Each combination was run twice in 27 randomly-generated 5x5 worlds with 2 cargo ships and 1-3 subs for 30 timesteps.

The average costs incurred can be seen in Figure 1-5. 30 particles outperforms 10 particles across the board, and after 30 world samples and a horizon of 3-5, any differences seem to just be random noise, so I settled on 30 particles, 30 world samples, and a horizon of 5 (circled in green in the figure) for further experimentation.

1.4.2 Exact vs Particle Belief States

To see if the particle belief state was accurately modeling the true belief, I implemented an exact belief state. I tested the exact belief state and particle belief states with 10, 30, 100,

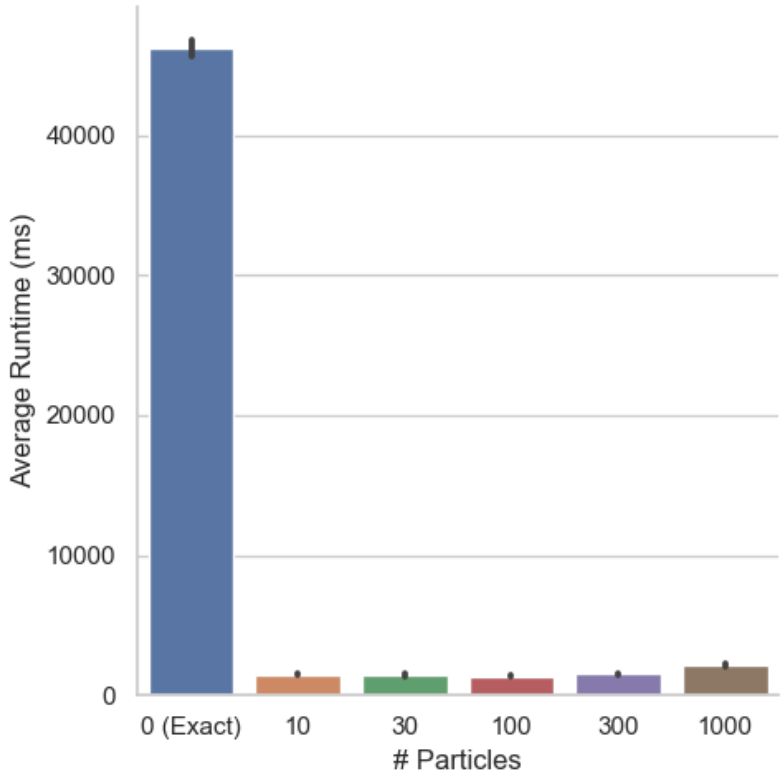

Figure 1-7: The average time hindsight optimization took to complete a test instance using an exact belief state or various numbers of particles.

300, and 1000 particles. Each variant was run twice in 100 randomly-generated worlds with 4 cargo ships and 1-3 subs for 30 timesteps.

Figure 1-6 shows a violin plot of the cost distribution for each variant. The white bar on each distribution is the mean with 95% confidence intervals and the dashed lines are the quartiles. According to this plot, even the 10-particle belief state has a lower average than the exact belief state. Application of the Kolmogorov–Smirnov test revealed that the exact belief's cost distribution is not significantly different from any of the others, but we would expect it to have a lower average because it tracks a probability for every possible state instead of just keeping an unweighted list of a few states. There are two possible explanations for this. The first is that there is a bug in my implementation of the exact belief. The second is that, because the agent's model of the submarines is inaccurate, perhaps calculating more accurate probabilities based on bad assumptions is actually worse than a more approximate approach such as particles. Either way, it seems that particles do work well. Increasing the number of particles beyond 30 does not seem to change much, so I stuck with 30 particles for my final experiment.

Figure 1-7 shows the time to complete a world for each of the variants. Every particle variant takes under 2 seconds, while the exact belief state takes over 40 seconds. It is clear that runtime is a disadvantage of the exact belief, and it has other disadvantages as well. Implementing it required precise knowledge of the inner workings of my domain, which is something that might not be available for a given domain. It can also only be used on small worlds, as 5x5 worlds with 4 cargo ships and 1-3 subs already had over 15 million states to track, so memory could become an issue with more complex worlds.

1.4.3 Comparative Performance

I compared the performance of my hindsight optimization planner (Hindsight) to the following agent strategies:

Static Does not move.

Random Chooses an action to take at random.

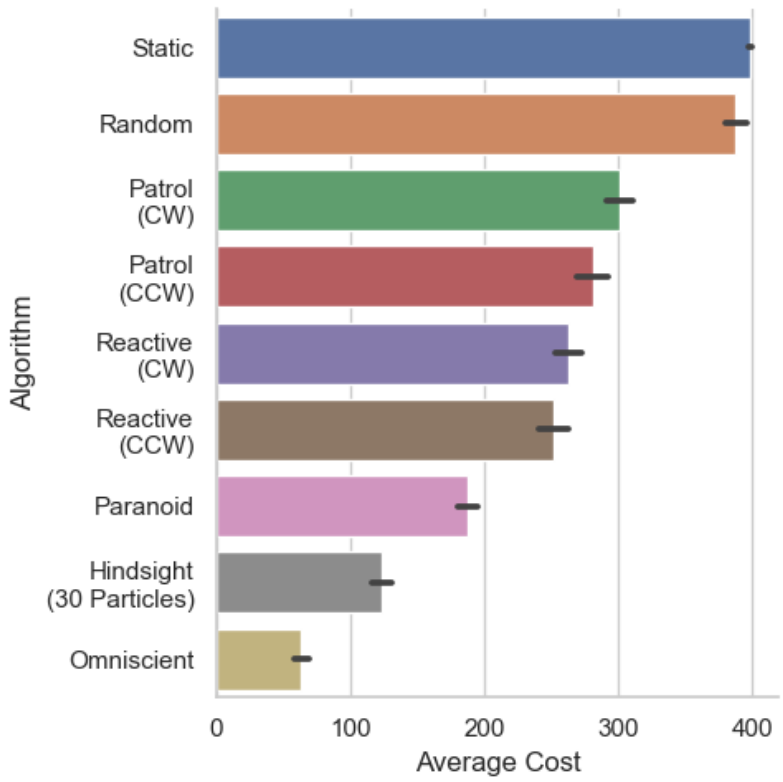

Figure 1-8: The average cost incurred by each algorithm.

Patrol Moves in a rectangle with size based on initial location (just like the cargo ships). There are two versions of this algorithm, CW and CCW, which determine which direction the agent will move in (clockwise or counterclockwise). Patrol is similar to the PLAN1 benchmark algorithm for ARTUE.

Reactive Like Patrol, except after a nearby cargo ship is attacked, moves to protect it and then updates its rectangular course based on its new location and its direction based on that cargo ship's direction. Reactive is similar to the REPLAN benchmark algorithm for ARTUE.

Paranoid Like Hindsight, but it samples random possible states based only on the current observation instead of maintaining a belief state (so it never learns where subs actually are).

Omniscient Uses the same planner as Hindsight (and so has a lookahead horizon) but knows the true state of the world (can see all subs) and uses that instead of hallucinating possible worlds.

Each algorithm was run for 30 time steps on 200 randomly-generated 7x7 worlds with 4 cargo ships and 1-3 subs. Hindsight used an unweighted particle belief state with 30 particles. Paranoid and Hindsight used sample sizes of 30 and horizons of 5 and Omniscient used a horizon of 5. 3 trials were done per world with each algorithm. The same seeds were used for each set of trials.

Figure 1-8 presents the mean cost incurred by each method, along with 95% confidence intervals. The results show that Hindsight beats all of the benchmark algorithms by a significant margin except Omniscient, which we would expect to represent an upper bound on achievable performance. The two versions of Patrol perform about the same, and so do the two versions of Reactive. This is not surprising, as the only difference between each version is starting direction, and the worlds are random, so neither starting direction should be better than the other on average. Sampling possible worlds and planning in them, even without updating belief state, is much more effective than the Static, Random, Patrol, and Reactive strategies, as evidenced by the large drop in cost from Reactive to Paranoid.

Hindsight exhibits emergent patrolling behavior, an example of which can be viewed here: `https://youtu.be/Gh6Ku05Y880`. It also displays implicit goal reasoning in an example where it believes that there are no submarines to begin with (and thus stays still to avoid the movement cost) but has to reevaluate its beliefs and "goals" when both cargo ships are simultaneously attacked: `https://youtu.be/FbqZVc9gCJg`.

1.5 Discussion

The results show that hindsight optimization works well on a small grid-based domain that shares some similarities with the TAO Sandbox domains used by Molineaux, Klenk, and Aha. The agent is never given the goal to patrol or a rule stipulating that it destroy a submarine when doing so would not put cargo ships in danger from other subs, and yet it acts in these ways. In other words, it displays goal reasoning without GDA.

Although Molineaux, Klenk, and Aha (2010) showed that their system did not perform as well when its goal reasoning component was removed, this provides only weak evidence that goal reasoning is useful. After all, removing the carburetor from an internal combustion engine does not prove that it is a necessary component — fuel injection or an electric motor might in fact be superior. Similarly, I expect that using a full-fledged planner would provide superior performance to a hand-tuned goal prioritization scheme, as only a planner can properly estimate the cost of achieving each possible outcome in the current context. One could call this goal reasoning, but the fact remains that planning-type reasoning is essential in action selection.

A limitation of this work is that the Navy Defense domain is simple, and the real world is not. Planning for a real autonomous Navy vessel would be much more complicated than planning on a grid, and in a real scenario the submarines would probably try to sink the Navy ship first, but in this game they only attack the Navy ship when they have no other choice. However, the TAO Sandbox is an approximation of reality, and Navy Defense is an approximation of the TAO Sandbox, so it serves its purpose.

A possible extension is to modify the Navy Defense domain by adding features from the

TAO Sandbox *Iceberg* scenario, such as allowing the agent to rescue people from sinking cargo ships, to show that the agent is able to display goal reasoning in more complex situations.

1.6 Summary

I add to prior arguments that planning can enable implicit goal reasoning without the goal-driven autonomy framework. To provide an example, I introduced the open-world POSG Navy Defense domain, which mimics some features of the TAO Sandbox, and then showed that a planner based on hindsight optimization can be used to find good online suboptimal solutions in Navy Defense scenarios, just as ARTUE can in the TAO Sandbox. With hindsight optimization, patrolling and goal reasoning behavior naturally emerge from the single goal of minimizing cost. This is a simpler and more general approach than ARTUE, as most of the reactive behavior in ARTUE is hard-coded. Therefore, this chapter provides evidence that general planners can be better solutions than specialized ones in POSGs. The next chapter will reinforce this idea by exploring general planning in a complex commercial POSG.

CHAPTER 2

Duelyst

Duelyst II (henceforth referred to simply as "Duelyst") is a two-player online collectible card game (CCG) that features a 9x5 grid board where unit cards can move around and fight. Each player's goal is to keep their General alive and kill the enemy General using the resources in their deck.

To my knowledge, Duelyst is an unexplored domain in AI research. Creating an AI to intelligently play Duelyst is a challenge because there is a large branching factor, several actions can be taken per turn within a 90-second limit, actions can be stochastic, and the opponent's deck and hand are unknown — it is a complex zero-sum POSG. The existing AI in Duelyst, called the "starter AI," is an expert-rule-based player that can only use a limited range of decks and is weak against humans. There are many ways in which a smarter AI that can play any deck could improve the player experience and introduce new content to the game. For instance, there is a "sandbox" mode where a player can test two of their decks against each other, but they have to play both decks themselves. I believe it would be more interesting and useful to fight a strong AI. There used to be boss battles in the game and there are plans for a roguelike mode (where the player battles through AI encounters), both of which could benefit from better AI. Currently, fighting the AI gives no rewards because it is too easy, but a strong AI could even be used in ranked games when few humans are online to speed up matchmaking times. In this chapter, I will describe general planners, including a modified version of MCTS, that are able to consistently beat the starter AI.

In the following sections, I shall discuss relevant work on games, describe the Duelyst domain and the starter AI, introduce my variant of MCTS, explain my static evaluator,

explore experimental results, and lastly consider limitations and possible extensions.

2.1 Related Work

Texas Hold'em Poker Zinkevich et al. (2007) introduce an algorithm called *counterfactual regret minimization* for finding approximate solutions to *extensive games with imperfect information* (i.e. POSGs) with up to 10^{12} game states. An extensive game with imperfect information can be modeled as a game tree and information sets for each player about the part of the state that they cannot observe. Chance is considered a player. The algorithm involves finding the ϵ-*Nash equilibrium* (where each player chooses roughly the best strategy for themselves) by minimizing *counterfactual regret*, a novel application of regret minimization that is akin to Bellman backups. *Regret* is essentially the utility missed out on by not playing an optimal strategy. Zinkevich et al. test their algorithm on two-player limited Texas Hold'em poker and beat champion algorithms.

I did not attempt to use counterfactual regret minimization for Duelyst, as finding an equilibrium (even an approximate one) would be difficult due to the large branching factor and several actions that each player can take in a turn. Sufficiently exploring a single ply is my current focus.

Magic: The Gathering Cowling, Ward, and Powley (2012) implement an expert-rule-based player for a simplified version of the popular trading card game *Magic: The Gathering*. *Magic* is a more complex version of Duelyst, although it does not have a grid board, but Cowling, Ward, and Powley (2012)'s version is simpler than Duelyst. They use an ensemble of determinized binary MCTS trees whose final action values get added together to consistently beat the expert-rule-based player. The binary trees (whether to take a certain action or not at each choice) are very fast.

This is an interesting approach and one that could be promising in Duelyst, but the binary trees are only used for playing cards in *Magic* and the choices are ordered by highest-cost card first, and it is unclear how to order Duelyst actions, since, unlike in *Magic*, playing cards and taking other actions are interleaved, not separate game phases.

Skat Furtak and Buro (2013) introduce *imperfect-information Monte Carlo* (IIMC) search, which is a PIMC search that uses PIMC to play out games instead of a deterministic planner (although the second-level PIMC searches use deterministic planners). They test IIMC in the card game Skat with favorable results, although it is very slow.

IIMC could theoretically be applied to Duelyst, but it would be too slow to be of any use, as the branching factor is too large for deterministic planning.

SparCraft Lelis (2017) introduces the *stratified strategy selection* (SSS) algorithm for controlling units in real-time strategy games. He implements a dynamic type system that partitions units into types based on certain features (like health, damage, and attack range) and SSS searches to find the optimal *script* to apply to all units of the same type, which determines what actions they will take. SSS+ does meta-reasoning about computation time to adjust the granularity of type system between calls. Lelis (2017) tests these algorithms in SparCraft, a simplified version of the popular real-time strategy game *StarCraft*. SSS assumes that the game is deterministic (which it is).

Despite the fact that Duelyst is a stochastic turn-based game rather than a deterministic real-time game, I believe that SSS+ could be an interesting algorithm to try, since Duelyst has units on a board with the same kind of features as appear in a real-time strategy game. However, this algorithm could only work for moving and attacking with units on the board, not taking other actions like playing or replacing cards.

Hearthstone Dockhorn et al. (2018) use a modified version of MCTS (see Figure 2-1) to play *Hearthstone*, a popular CCG that is very similar to Duelyst, except that it does not have the grid board. They run MCTS for the agent, rolling out until the end of the turn instead of the end of the game, then select the n best end-of-turn nodes reached during rollouts and run MCTS on those for the opponent, then repeat this process to simulate the agent's next turn from the n best ending opponent states. In order to simulate the opponent, they predict that the opponent's deck is one of 20 preset options (whichever is closest to what they have played) and choose a hand from that using a bigram model of card co-occurrences. In this way, they are able to look ahead 3 plies while planning, though only

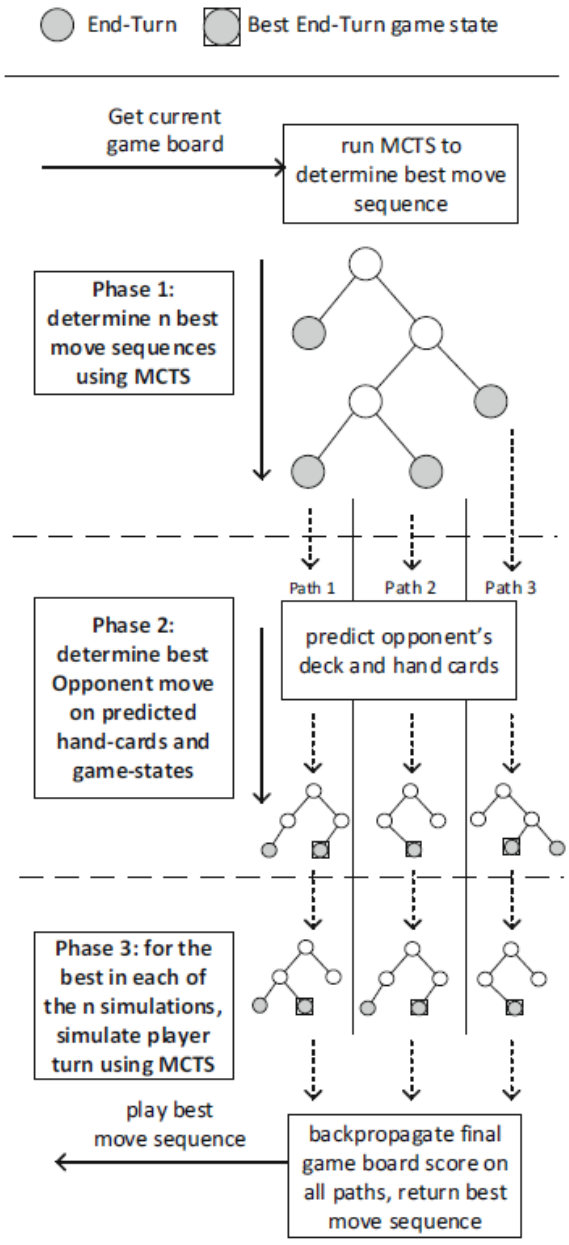

Figure 2-1: The modified MCTS used for Hearthstone by Dockhorn et al. (2018). Image from Dockhorn et al. (2018).

for a few simulations. The evaluations of the final states (3 plies ahead) are backpropagated to the current-ply search tree. The algorithm returns the best sequence of actions from the search tree (until the frontier) instead of just the best next action to take. Rollouts are done greedily using a static evaluator.

Since *Hearthstone* is so similar to Duelyst, this approach would likely work for Duelyst as well, although it is very slow. I borrowed the idea of rolling out until the end of the turn using a greedy policy for my algorithms (as will be discussed in Section 2.4) but do not yet explore multiple plies.

Settlers of Catan Dobre and Lascarides (2018) use POMCP to play *Settlers of Catan*, a complex 4-player POSG. They cluster actions into types — such as road building, city building, and trading — and learn a preference distribution over those types from game data. This distribution is used to influence which actions get selected during tree search (the policy is PUCT rather than UCT) and rollouts (an action type is sampled first, then an action from that type).

Although *Settlers of Catan* and Duelyst are very different games, they both have a board, hidden hands of cards, and stochastic actions. Duelyst also has actions that can be easily clustered into types, and I incorporated this idea into my MCTS variant (as will be discussed in Section 2.4.1). POMCP would be a good approach for a multi-ply search, but for a single ply, MCTS is sufficient.

Cyber Security Mern et al. (2020) introduce the *Bayesian optimized Monte Carlo planning algorithm* (BOMCP) algorithm, which is POMCP with belief nodes instead of observation nodes. Vectorized versions of actions and beliefs are required for this method because Bayesian optimization is used to choose between newly created action nodes based on a Gaussian process constructed from past beliefs and actions. This is meant to help the planner learn what actions are good generally and deal with large action spaces. Mern et al. test BOMCP on a few domains, including a cyber security game where the agent must stop a stochastic virus adversary from infecting nodes in a LAN.

The cyber security domain has no relevance to Duelyst besides being a POSG, but

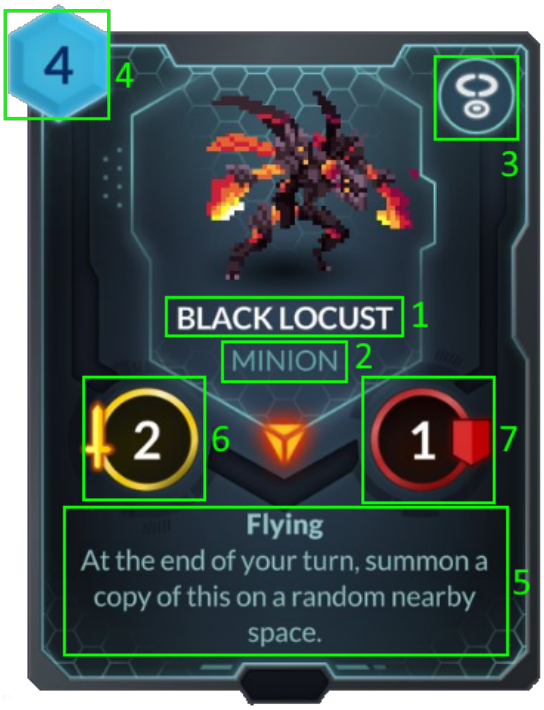

Figure 2-2: The anatomy of a card. 1 is the name, 2 is the type (which can be General, minion, spell, or artifact), 3 is the faction (of which there are six, plus neutral, as this card is), 4 is the mana cost, and 5 are the abilities. 6 and 7 only exist on General and minion cards, and they are the Attack and Health, respectively. Bolded abilities like Flying are known as *keywords*, and are reused across many cards. Black Locust is an example of a card with a stochastic ability.

BOMCP could help deal with Duelyst's large branching factor. It may not be difficult to vectorize actions and beliefs, considering that actions are made up of a type and a set of board coordinates and beliefs just have to be the opponent's deck and hand, and those are already represented by lists of integers.

2.2 The Duelyst Domain

In CCGs like Duelyst, players accumulate cards, build decks with them, and then choose a deck to fight another player (human or AI) with. Deck building and selection are done

Figure 2-3: A screenshot of *Duelyst II*. 1 is the 9x5 grid board, 2 is Player 1's General, 3 is Player 1's mana (current / total), 4 is Player 1's hand, and 5 is the artifact with three durability that Player 1's General has equipped. Player 1's deck size can be seen in the bottom-left corner. Player 2's hand size and mana can be seen in the top-right corner. Mousing over the opponent's profile picture reveals their deck size.

before the matches and are outside the scope of this research.

In a Duelyst match, two players do battle with 39-card decks on a 9x5 grid board. There is a maximum hand size of 6. Each player controls a General and their goal is to kill the opponent's General (by reducing that General's Health to 0). To do this, they can play minion, spell, and artifact cards from their hand by spending mana. Minions are played on the board and can move and attack enemy units (enemy units are the opponent's minions and General). Spells have an effect when played. Artifacts boost the power of a player's General until they are destroyed (they have three durability and lose one every time the General is hurt). Duelyst is partially-observable because each player cannot see their opponent's deck or hand.

Duelyst has a discrete but very large state space. There are 6 factions, and each faction has 34 cards — 3 artifacts, 14 minions, and 17 spells. There are 112 neutral cards, all minions. A player's deck can be made up of cards from one faction and neutral cards and can contain up to 3 copies of the same card, which means there are $\binom{(34+112)\times 3}{39} \approx 8.93\times 10^{55}$ possible decks for a given faction. There are also 2 tiles, which go under minions on spaces, and 26 token cards that can only be created during matches (not added to decks). 45 units and tiles can be on the board at once, 6 cards in each player's hand, and potentially more than 39 cards in each player's deck (because some cards add cards to a player's deck). Units on the board can have Attack ranging from 0 to 99 and Health ranging from 1 to 99 (both integers) and can also have an unlimited number of additional abilities granted to them by cards or lose their abilities entirely. All of this makes the state space enormous.

Duelyst has a discrete action space that can be relatively small or very large depending on how many cards are in the player's hand and how many units are on the board. It is not uncommon to have a branching factor of over 100 at some points during the game. In addition, several actions can be taken in a turn within a 90-second time limit and some actions have stochastic outcomes. Obviously, drawing cards is stochastic, but some cards have stochastic abilities as well.

Figure 2-2 explains the parts of a card and Figure 2-3 explains a screenshot from a

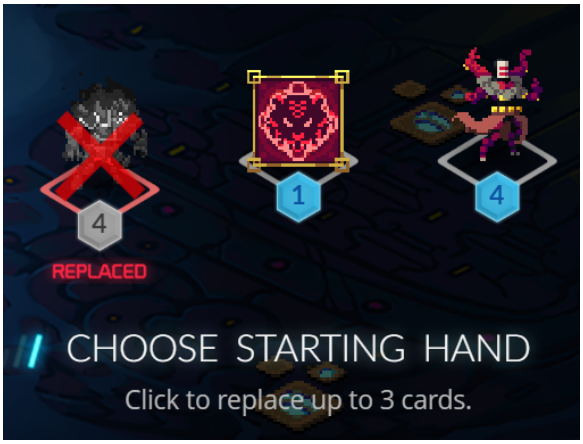

Figure 2-4: The mulligan phase at the beginning of a game.

match. In this section, the mechanics of the game will be fleshed out.

2.2.1 Deck Composition

Duelyst decks are technically 40 cards, but as one of those cards is the General and all Generals are the same aside from their appearances and always start out on the board, decks can be considered 39 cards. A deck is made up of cards from one of six factions — Lyonar, Songhai, Vetruvian, Abyssian, Magmar, or Vanar — and/or neutral cards. Up to 3 copies of a specific card can be in a deck, so decks can contain anywhere from 13 to 39 unique cards (a deck composed entirely of unique cards in a CCG is called a "singleton" or "highlander" deck).

2.2.2 Beginning a Match

In the beginning of a match, you learn which of the six factions your opponent is playing and whether you are going first or second. Then, you draw three cards from your deck and have a limited amount of time to mark any number of these cards for replacement. You draw cards equal to the number of cards you marked and then shuffle the marked cards back into your deck. The remaining three cards are your starting hand. This is known as a *mulligan* and is a common feature in CCGs. Figure 2-4 shows the mulligan phase.

Figure 2-5: The initial game state.

Mulligans happen simultaneously for both players, but after that the game is turn-based. If you are going first, you start with 2 max mana and your General starts on the left side of the board; if you are going second, you start with 3 max mana and your General starts on the right side of the board. The initial board state can be seen in Figure 2-5. The blue orbs are mana spring tiles (also called mana orbs) that grant 1 mana to the player who first gets a minion or General on that space. Mana orbs are meant to incentivize players to move towards each other and engage in combat instead of building up armies on their respective sides.

2.2.3 Ending a Match

The player who kills the other player's General first wins. If both Generals die at the same time, the match is a draw. Generals start with 25 Health and die when they reach 0 or less Health, like all units.

Figure 2-6: The glowing yellow spaces are the actions available for playing a specific minion card. There are 15 in this case with only 2 allied units on the board. Playing this minion will only introduce more options for placement of the next minion.

2.2.4 Turn Structure

At the beginning of your turn, you gain 1 max mana (unless it is your first turn) and then your mana is set equal to your max mana. You then have 90 seconds to perform actions. The types of actions are:

1. **Replace a card** in your hand. The card gets shuffled into your deck and you draw a different card (if possible). This can only be done once per turn.

2. **Play a card** from your hand. You must have enough mana to play the card and some cards also require a specific kind of target to be on the board in order to be played. Once the card is played, you lose mana equal to its mana cost.

3. **Move an allied unit** on the board. Units can only be moved once per turn and only if they have not attacked this turn and were not played this turn.

4. **Attack with an allied unit** on the board. Units can only attack once per turn and only if they were not played this turn.

5. **Follow-up** a previous action. Follow-up actions only become available as the result of playing certain cards. An example is a picking multiple targets for a card's effect.

6. **End your turn**. Pass play to the opponent. At the end of your turn, you draw up to 2 cards. If your hand is full, you cannot draw; if your deck is empty, your General takes 2 damage per card that would have been drawn.

These action types each represent several actions that may or may not be available in a given state. Actions can be taken in any order, and taking an action often opens up more possibilities. The number of possible actions depends on the number of units on the board and the number of cards in hand. Artifacts can be played anywhere on the board, spells often require a specific target (i.e. an enemy minion) to cast, leading to potentially numerous options, and minions can be played anywhere around an allied unit, which means up to eight different actions per allied unit. An example of the actions available when playing a minion is shown in Figure 2-6. Minions sometimes have abilities that trigger as they are being played and require a target (follow-ups), which makes playing them require two actions (placement and targeting). Units can move up to 2 spaces orthogonally or 1 space diagonally unless impeded by an enemy unit (up to 12 possibilities), but units with the Flying keyword (see Figure 2-2) can move to any open space on the board, resulting in up to 42 possibilities. Units can attack any nearby enemy unit (up to 8 possibilities), but units with the Ranged keyword can attack any enemy unit. All of this is why the branching factor of Duelyst can be quite large — there are often over 100 possible actions.

2.2.5 Combat

Combat between units is perhaps the most important part of playing Duelyst; it is usually the way games are won and lost. When an allied unit attacks an enemy unit, it subtracts its Attack value from the enemy's Health and then the enemy counterattacks and subtracts

Figure 2-7: Before and after of an attack action.

its Attack value from the ally's Health. An example of this is shown in Figure 2-7. The General (on the left) that is attacking the enemy unit (on the right) becomes *exhausted* after attacking, meaning that it cannot move or attack again until the player's next turn. After the attack and counterattack, if any of the fighting units are at 0 or less Health, they die and are removed from the board.

2.3 The Starter AI

The starter AI is an expert-rule-based player. This means that it follows a set of hard-coded instructions about how to play the game that were created using expert-level domain knowledge. These instructions lead to rigid, predictable behavior. For instance, the starter AI will always replace a card from its hand as its first action, even though it may be beneficial in some cases to wait or not replace a card at all. To be able to handle the large state space of Duelyst, most of the instructions are somewhat general, like to play the most expensive card in hand first. However, the starter AI does rely on card *intents*, which are rules for how to play specific cards effectively. The AI has intents for most of the cards in the starter decks (the decks players first unlock), which is why it is called the starter AI.

A proficient human player can consistently beat the starter AI with no trouble. Currently it is only being used for training new players to play the game. A player who is new to Duelyst might lose against it once or twice, but it is unlikely to pose much of a challenge

once they get a feel for the game.

Another downside of this approach is that it is inflexible. Cards are often changed in CCGs to "balance" them (make them more fair), and unless the associated card intents are updated too, the starter AI will have the wrong idea about how to play these cards intelligently. In addition, there are far more cards in Duelyst than the AI has intents for — and more will be added in expansion updates — and it does not know how to play well with these cards.

Advantages of the starter AI are that it makes decent decisions (at least with the starter decks) very quickly and takes very little, if any, memory to do so. Here is an interactive demo of me playing against the starter AI: `https://api.duelyst2.com/replay?replayId=-NUPAjAe-Tn7zAzpphVU`. I am the player named You and the AI is named Argeon Highmayne. I am using the Songhai starter deck and the AI is using the Lyonar starter deck. Most of its plays are reasonable, but even so, I am able to beat it without difficulty.

2.4 Modified MCTS

My algorithm for Duelyst is called *one-ply MCTS for CCGs*. It is MCTS but with three modifications:

1. Rollouts only go until the end of the turn instead of the end of the game, because the opponent's deck and hand are unknown. Rollouts either take random actions or do hill-climbing, greedily choosing the action that leads to the highest-valued state (according to the static evaluator). The latter is the approach taken by Dockhorn et al. (2018).

2. Instead of action nodes being generated as children to a state node, an action tree can optionally be generated instead. Action trees are explained below.

3. There is some code specific to dealing with the simultaneous mulligans that Duelyst and other CCGs have. If this minor change is removed, the algorithm becomes general to all turn-based games where only exploring the current ply is desired.

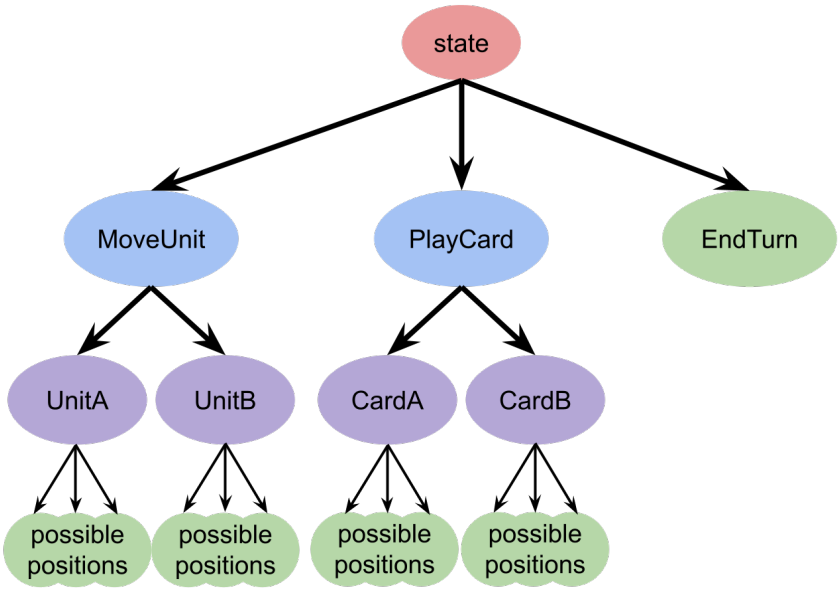

Figure 2-8: An action tree for a particular state.

There are parameters for the rollout policy to use (random or greedy), the time to spend searching for each action, whether or not to use action trees, and whether to choose an action by its value or its visitation count, as well as the default MCTS exploration parameter C.

I will first explain action trees and then the algorithm in detail.

2.4.1 Action Trees

Due to the large branching factor that Duelyst has, I wondered if there would be enough time to run sufficient rollouts to determine a smart action. Action trees are meant to help the agent cope with large branching factors by grouping information about similar actions.

Inspired by Dobre and Lascarides (2018)'s use of action types to guide search, action types are an explicit part of the MCTS search tree — the possible actions at a particular state are grouped into an *action tree*. Duelyst's action space can be neatly factored into seven action types: Mulligan, MoveUnit, AttackWithUnit, PlayCard, ReplaceCard, EndTurn, and Followup (a Followup action can occur after a PlayCard action or another Followup action). For the MoveUnit, AttackWithUnit, and PlayCard action types, they can

Algorithm 3 One-Ply MCTS for CCGs(G, P)

```
1: T ← empty tree
2: // Take actions until the turn is over
3: while G.isPlayerTurn(P) and G.timeLeftInTurn() > TimePerAction do
4:     a ← Search(G, P, T)
5:     G.takeAction(a)
```

be further factored (in the next layer of the action tree) into *action sources*, or the unit or card that the agent is going to perform that action with (e.g. MoveUnit could branch into UnitA and UnitB). Underneath action types or action sources are ground actions, which are fully described actions that can be executed in the state (e.g. where to move UnitB). The EndTurn action type is unique in also being a ground action. Each action type and action source node has a visitation count and stored value, just like the other nodes in the search tree, and choosing an action consists of choosing an action type, (potentially) an action source, and then a ground action.

Figure 2-8 shows an example of an action tree for a particular state. Action types are blue, action sources are purple, and ground actions are green.

2.4.2 The Algorithm

The high-level procedure for one-ply MCTS for CCGs is outlined in Algorithm 3. It takes the game object (G) and the AI player's ID (P) and repeatedly searches for and takes an action until that player's turn is over (lines 3-5). G.isPlayerTurn is assumed to evaluate to true for either player during the simultaneous mulligan phase at the beginning of the game, unless that player has already taken a mulligan action (chose which cards in their starting hand to replace).

The search procedure is outlined in Algorithm 4. It repeatedly simulates from the root of the tree, following the tree policy while in the search tree, then doing a rollout on the new node that is added, then backpropagating the value back up the tree (lines 16-19). It then returns the best action from the root by choosing the highest-valued child node until a ground action is reached (lines 22-23). There is a parameter to choose the most-visited

Algorithm 4 SEARCH(G, P, T)

```
6:  // Create or set root of tree
7:  if T.root = null then
8:      T.root ← NODE(G.curState)
9:  else  // The root is an action
10:     for n ← T.root.children do
11:         if n.state = G.curState then
12:             T.root ← n
13:     if T.root.isAction then
14:         T.root ← NODE(G.curState)
15: // Repeatedly simulate from the root
16: while TimePerAction is not out do
17:     n ← TREEPOLICY(G, P, T.root)
18:     v ← ROLLOUT(G, P, n.state)
19:     BACKPROPAGATE(n, v)
20: // Find and return best action
21: n ← T.root
22: while !n.isAction do
23:     n ← n's max-valued child   // Bypasses action types and sources
24: T.root ← n
25: return n.action
```

child node instead (not shown). The tree is reset between turns (line 1) but the relevant parts are saved between actions in the same turn, assuming the last executed action led to a state that was discovered during search (line 12).

The tree policy is outlined in Algorithm 5. It explores the search tree using the UCT algorithm to select action types, action sources, and ground actions (line 34), and when it reaches a state that is not in the tree, it adds a node containing that state to the search tree and returns it (lines 46-48). It can also add an action tree to the search tree on the way to a new state (line 30). There is a parameter to just generate ground action nodes instead (not shown). Newly created nodes have V = 0 and N = 0, where V is the estimated value of the node and N is the visitation count. EndTurn nodes will not be explored twice if the action they represent is deterministic (line 33), and if a stochastic EndTurn node is explored again and leads to a previously discovered state, that state will be returned anyway (line

Algorithm 5 TreePolicy(G, P, node)

```
26: n ← node
27: s ← n.state
28: // Generate action tree from node if it doesn't already exist
29: if n.children = ∅ then
30:     n.genActionTree()
31: // Traverse action tree and take an action
32: while !n.isAction do
33:     children ← n.children - (EndTurn if it exists, its N = 1, its action is deterministic,
        and it is not the only option)
34:     act ← argmax_{child∈children} ( child.V + C sqrt(log n.N / child.N)  if child.N > 0,  else ∞ )
35:     n ← n.children[act]
36: s' ← G.simAction(s, n.action)
37: // If going second, skip opponent's turn after mulligan to search in first turn
38: if n.parent = Mulligan and G.playerTurn(s') != P then
39:     s' ← G.simAction(s', EndTurn)
40: // If resulting state does not exist in tree, add new node and return it
41: // Otherwise, keep traversing tree
42: for child ← n.children do
43:     if child.state = s' then
44:         n ← child
45: if n.isAction then
46:     newNode ← Node(s')
47:     n.children.add(newNode)
48:     return newNode
49: else if n.parent = EndTurn or G.gameIsOver(n.state) then
50:     return n   // Prevent traversal of states in opponent's turn or beyond end of game
        by returning existing node
51: else
52:     return TreePolicy(G, P, n)
```

50) instead of continuing to search into the opponent's turn. In this case, no nodes will be added to the search tree. During the mulligan phase, the AI will look ahead into its first turn in order to inform its card replace decisions. If the AI is going second, it will ignore the opponent's first turn (pretend that the opponent does nothing) in order to continue searching in its first turn (line 39).

Algorithm 6 ROLLOUT(G, P, s)

```
53: // If the player's turn just ended, return the predicted value of this state
54: if G.playerTurn(s) != P then
55:     return STATICEVAL(s)
56: // Greedily select action to take based on static evaluations
57: nextStates ← [G.simAction(s, a) for each a ← G.actions(s)]
58: a ← argmax_i(STATICEVAL(nextStates_i))
59: // Continue rollout from next state
60: s' = nextStates[a]
61: return ROLLOUT(G, P, s')
```

The rollout policy is outlined in Algorithm 6. Until the player's turn is over, it tries each action, applies the static evaluator to the resulting states, and chooses the action with the highest estimated value to continue rolling out with. There is a parameter to choose a random action instead (not shown). It returns the value of the state after the EndTurn action — the first state in the opponent's next turn. This is because there are many unit abilities that trigger at the end of the turn that should be accounted for in the evaluation.

When backpropagating values (line 19) from new leaf nodes up the tree to the root, the max of child values is taken at all nodes except ground action nodes, which take the mean of child values. This is because action nodes are like chance nodes in expectiminimax — different states can result from the same action due to the stochastic nature of the game.

A small enhancement (not shown) is if there is only one action (which would have to be an end turn action or a follow-up action), it is chosen immediately instead of doing the normal procedure.

2.5 Static Evaluator

The static evaluator I use for my algorithms estimates the value of a state using a set of simple rules. A win state has a value of 10,000 multiplied by the remaining health of the winner's General (in order to differentiate end-game states), a lose state is the same but with a negative value, and any other state's value is the difference in values of both players'

units. A unit's value is composed of two parts: an intrinsic value plus a positional value.

The intrinsic value for a General is the General's Attack plus 3 times the General's Health (to promote the ultimate objectives of staying alive and killing the enemy General). Finding intrinsic value for a minion is a bit more complex because minions have a wide range of abilities, some which are much more powerful than others. Luckily, minions also come with a built-in estimator of value: their mana cost. However, just using the mana cost as value is not enough, because minions have Attack and Health that can be modified, but their original Attack and Health are factored into their mana cost. My approach is to calculate the value of the minion's abilities and then add it to their combined current Attack and Health to get their intrinsic value.

To estimate the value of a minion's abilities, I subtract their combined original Attack and Health from their original total value. Original total value is only based on mana cost, but I have to get it in terms of "stat points" (i.e. 1 point of value should equal 1 Health or 1 Attack) so a minion with the highest stats (combined Attack and Health) for its cost has an ability value of 0. Many minions in the starter decks have no abilities, and these minions' stats act as an upper bound for other minions of the same mana cost (adding abilities subtracts stats). Since Duelyst games are supposed to be "lightning fast," stats do not increase linearly with mana cost (i.e. a 2-mana minion can have 5 stat points and a 3-mana minion can have 7 stat points but a 6-mana minion can have 16 stat points). I found that the following formula does a fairly good job of estimating a minion's original total value in terms of stat points:

$$originalTotalValue = \lfloor originalManaCost^{1.4} + 4 \rfloor$$

Some minions have what are called *Opening Gambit* abilities, which trigger when they are played. For minions with Opening Gambits, I say that their ability value is 0 instead of doing the above calculations. The reason for this is that nearly all minions with an Opening Gambit have no other abilities, and since Opening Gambits, unlike other abilities, are not persistent, they should not factor into a minion's value.

A unit's positional value is based on proximity to mana orbs and killable enemies. The

unit gets a penalty equal to the Manhattan distance it is away from the nearest active mana orb (encouraging moving towards the mana orbs in the beginning of the game). It also gets +1 value for each enemy minion within a 2-space radius it could kill and +100 value if it could kill the enemy General in that radius. If it is close to the enemy General but cannot kill them, it still gets +value equal to its Attack (potential damage to the enemy General).

It may seem strange that I am not considering cards in players' hands when calculating the value of a state in a card game. There are two reasons for this. The first is that, while cards in the agent's hand could be evaluated much as I do for units on the board, cards in the opponent's hand are hidden, so this would be one-sided. The second is that I tried giving each card a fixed value and subtracting the values of both players' hands as part of the static evaluation, but this ended up leading to bad plays because ending the turn would increase the value of the state by two cards' worth. Because players draw two cards every turn and have a maximum hand size of six, it is usually best to play as many cards as possible anyway.

2.6 Experimental Results

I conducted three experiments in the Duelyst domain. The first was to tune MCTS' parameters, the second was to compare the performance of MCTS to that of baseline algorithms, and the third was to see if going first in Duelyst gives the player an advantage. For all experiments, both AI players use the same deck (the Lyonar starter deck, the first deck players unlock) and the 90-second turn timer is ignored. I shall now delve into each of these experiments and their results.

2.6.1 Tuning MCTS

In order to see the effect of action trees, random or greedy rollouts, and choosing actions by value or visits on the performance of MCTS against the starter AI, I ran 1,000 games with MCTS using each combination of possible arguments. In every case, MCTS had 15 seconds to choose each action and C was set to 10.

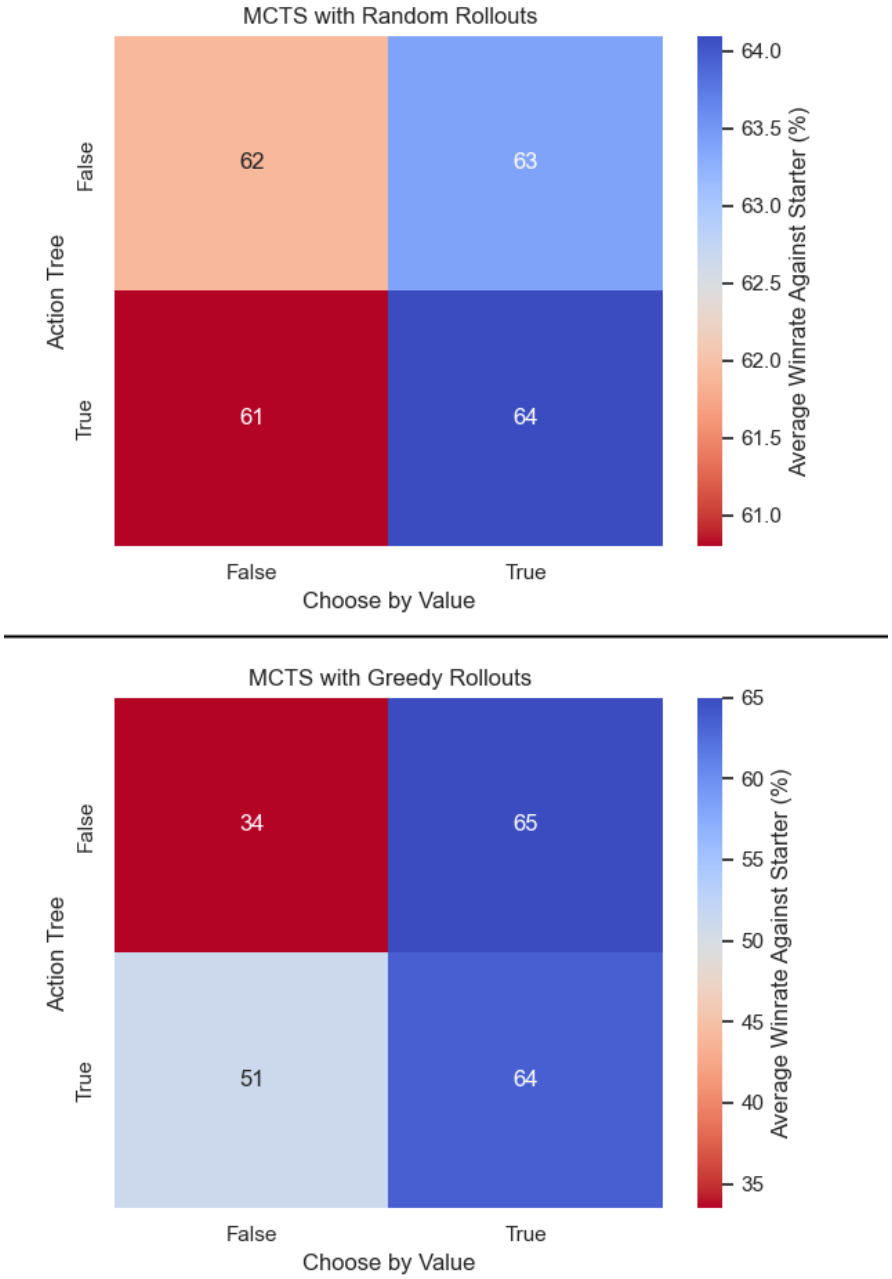

Figure 2-9: Heatmaps showing the average winrate MCTS had against the starter AI with different parameter values.

The average winrate for each combination is shown in Figure 2-9. With random rollouts, choosing by value is slightly better than choosing by visits, and using an action tree or not does not seem to make a difference. With greedy rollouts, choosing by value is significantly better than choosing by visits, and action trees only make a difference when choosing by visits — but it is a significant difference. When choosing by value, both random and greedy rollouts perform about the same. This is likely because random rollouts are faster and so more rollouts are possible, which increases knowledge about what actions are valuable even if the estimates start out inaccurate, but greedy rollouts get more accurate estimates of value with fewer rollouts, which amounts to the same thing. The fact that action trees only help when choosing by visits and using greedy rollouts may be for the same reason — because fewer greedy rollouts happen than random rollouts, perhaps the visitation counts of actions do not vary much even though the values do, but when actions are factored into trees there are fewer top-level choices and so the visitation counts vary more from the same number of rollouts, helping to choose better actions more of the time.

Every combination except greedy rollouts choosing by visits with no action trees beats the starter AI consistently (a greater than 50% winrate). Greedy rollouts choosing by visits with action trees is on the edge at 51% winrate, but the other successful methods are all above 60% winrate, with the highest being 65%.

2.6.2 Comparative Performance

In order to see if a MCTS-based approach was necessary to consistently beat the starter AI, I created the following baseline algorithms:

Random Takes a random action.

Hillclimb Simulates trying each action and chooses the one that leads to the highest-valued state (1-step lookahead).

Rollout Simulates trying each action, performs a rollout (random or greedy) from the resulting state, and chooses the action that led to the highest-valued end-of-turn state.

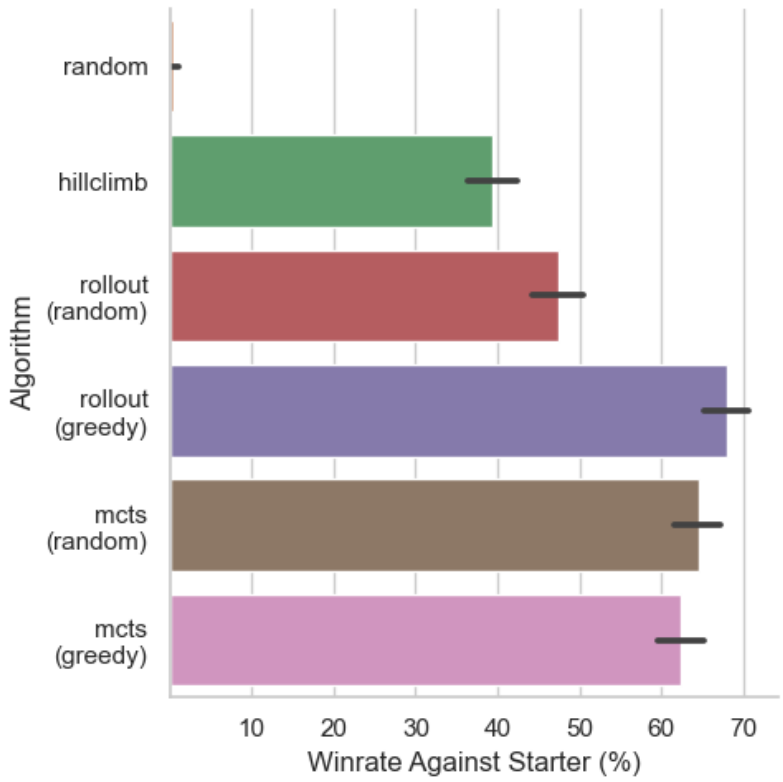

Figure 2-10: The winrate of each algorithm against the starter AI.

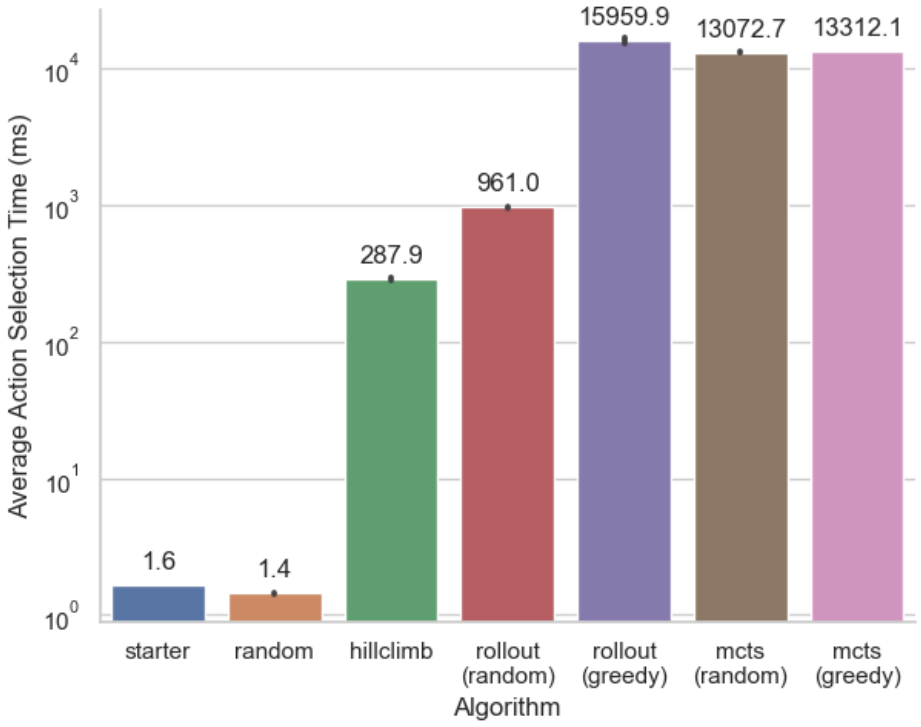

Figure 2-11: The average time (in milliseconds) that each algorithm took to select an action.

I ran Random, Hillclimb, Rollout with random and greedy rollouts, and MCTS with random and greedy rollouts against the starter AI 1,000 times each. Both MCTS versions had 15 seconds to choose each action, chose actions by value, and used a C of 10. MCTS (Random) did not use action trees but MCTS (Greedy) did.

The winrate of each algorithm is shown in Figure 2-10. Random, Hillclimb, and Rollout (Random) are not able to beat the starter AI consistently, but, as expected, Rollout (Random) outperforms Hillclimb, which outperforms Random (by a large margin). Both MCTS versions have similar winrates over 60%, as in the last experiment. What is unexpected is that Rollout (Greedy) outperforms MCTS (although not significantly), with a 68% winrate.

Figure 2-11 shows the average selection time of each algorithm, including the starter AI. Random is naturally the fastest, with the starter AI a close second. Rollout (Greedy) is the slowest, with an average of 16 seconds per action. Both MCTS methods take around 13 seconds on average instead of 15 because the algorithm immediately chooses an action when there is only one instead of searching for 15 seconds first. This plot shows that there is a price to pay for superior performance using a general approach — the algorithms that consistently beat the starter AI are very slow. However, even the other baseline algorithms are slower than they should be. The reason for this is that copying a Duelyst state is extremely slow, which will be explained in Section 2.7.

2.6.3 Is Going First Advantageous?

In many games, such as tic-tac-toe and chess, it is advantageous to go first. In order to find out if this was the case in Duelyst, too, I used the results of the previous experiment to create Figure 2-12, which shows the winrate of each algorithm going first and second. For every algorithm, going *second* actually leads to a higher winrate, and in most cases, the difference is significant. To offset the advantage of playing first, player 2 starts with an extra mana to begin with and also starts within movement distance of one of the three mana orbs (see Figure 2-5), allowing for stronger first-turn plays than player 1 and access to stronger cards sooner in general. This makes it easier for agents playing simple strategies

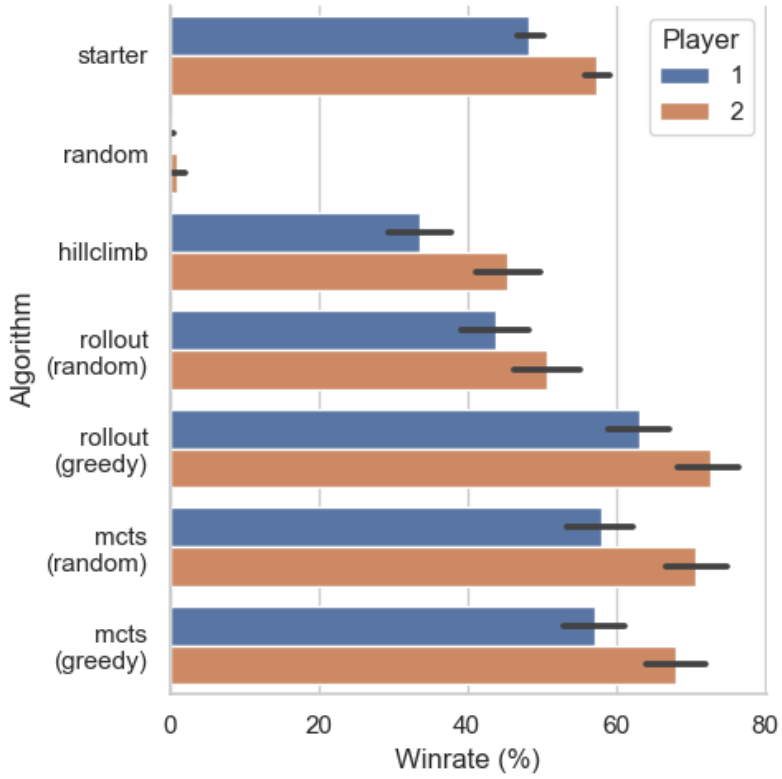

Figure 2-12: The winrate of each algorithm playing first and second.

(the only ones possible with the starter decks) to win when going second. However, this does not necessarily mean that going second is always advantageous when other decks are used that allow for more advanced strategies.

2.7 Discussion

The results show that a general planner even simpler than MCTS can consistently beat the specialized starter AI, although not in a reasonable time. However, I worked in the actual *Duelyst II* codebase, which is written in JavaScript, and this had some unique challenges. Originally, there was no equality comparator for game states, which is necessary for MCTS, so I had to implement one myself. The biggest problem is that the GameSession object holds not only the current state but the entire game history, and the existing copy function serializes it to a JSON and then deserializes it, which is incredibly slow. I would like to implement a more efficient method of state copying that leaves out or points to information that does not need to be copied for simulation purposes. This could speed up all of my algorithms (except Random), which is important for having any of the successful ones be viable options in the actual game (players do not want to wait 16 seconds for the AI to make a move).

A limitation of this work is that I did not have enough time to test different values of C or different time limits for MCTS. It would be interesting to see what the cut-off in performance is and if searching for less than 15 seconds can still yield intelligent actions.

Another limitation is that the Lyonar starter deck, which all algorithms used, does not contain any cards with stochastic abilities. Replace and end turn actions are still stochastic, but using a starter deck that introduces more randomness (such as the Abyssian starter deck) might give a better indicator of how well these algorithms actually perform.

A possible extension that I have already started experimenting with is multi-ply planners; that is, algorithms that maintain a belief about the opponent's deck and hand and simulate their turn. Preliminary results in this direction are not promising, but I have some ideas that I want to try, including *counterplay planning* (named after the original *Duelyst*'s

creator Counterplay Games), which would plan for the opponent as if they had two turns in a row and then try to counter their most damaging potential actions.

Another possible extension is to take more advantage of the fact that Duelyst actions are marked as containing randomness or not to avoid simulating deterministic actions that have already been explored during MCTS, which should speed up the search.

2.8 Summary

I introduced a MCTS variant, one-ply MCTS for CCGs, and the notion of factoring actions into action trees based on their types and sources. I also developed a simple static evaluator for Duelyst. Then I pitted my MCTS and some baseline algorithms against the expert-rule-based starter AI in the complex commercial POSG *Duelyst II* and found that not only MCTS but also one of the baselines could consistently beat the starter AI. Therefore, this chapter provides evidence that general planners can be better solutions than specialized ones in POSGs.

CHAPTER 3

Conclusion

Partially-observable stochastic games (POSGs), which have both hidden state and uncertain action outcomes, are complex domains that are usually intractable to solve optimally, and thus require suboptimal online planners that can find good solutions.

General-purpose planning algorithms are simpler to implement and understand than specialized algorithms made for a certain domain, and they require far less domain knowledge. I explored general-purpose planners in two POSGs.

In Navy simulation games, ARTUE, an expert-rule-based algorithm built on the goal-driven autonomy framework, performs well, but so does hindsight optimization, a simple general planner using minimal domain knowledge in the form of a belief state. Hindsight optimization displays emergent intelligent behavior such as patrolling and goal reasoning despite only having the single goal to minimize cost.

In the card game Duelyst, the starter AI, an expert-rule-based algorithm, is mediocre against humans, and two simple general planners using minimal domain knowledge in the form of a static evaluator are able to consistently beat it. Domain knowledge about action types can be leveraged in MCTS and is helpful in some situations. These algorithms avoid the partially-observable aspect of the game by planning only in the current ply, but approaches that simulate the opponent could be even stronger.

General-purpose planning algorithms can perform just as well or better than specialized algorithms, even in complex domains like POSGs.

Bibliography

Burns, E.; Benton, J.; Ruml, W.; Yoon, S.; and Do, M. 2012. Anticipatory On-line Planning. In *Proceedings of ICAPS-12*.

Chong, E.; Givan, R.; and Chang, H. S. 2000. A framework for simulation-based network control via hindsight optimization. In *Proceedings of the 39th IEEE Conference on Decision and Control*.

Cowling, P. I.; Ward, C. D.; and Powley, E. J. 2012. Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. *IEEE Transactions on Computational Intelligence and AI in Games*.

Dobre, M.; and Lascarides, A. 2018. POMCP with Human Preferences in Settlers of Catan. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Dockhorn, A.; Frick, M.; Akkaya, Ü.; and Kruse, R. 2018. Predicting Opponent Moves for Improving Hearthstone AI. In *Information Processing and Management of Uncertainty in Knowledge-Based Systems. Theory and Foundations*.

Fox, D.; Hightower, J.; Liao, L.; Schulz, D.; and Borriello, G. 2003. Bayesian filtering for location estimation. *IEEE Pervasive Computing*.

Furtak, T.; and Buro, M. 2013. Recursive Monte Carlo search for imperfect information games. In *Proceedings of CIG-13)*.

Gelly, S.; Kocsis, L.; Schoenauer, M.; Sebag, M.; Silver, D.; Szepesvári, C.; and Teytaud, O. 2012. The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM*.

Ginsberg, M. L. 2011. GIB: Imperfect Information in a Computationally Challenging Game. *CoRR*.

Kiesel, S.; Burns, E.; Ruml, W.; Benton, J.; and Kreimendahl, F. 2013. Open World Planning for Robots via Hindsight Optimization. In *Proceedings of the ICAPS-13 PlanRob Workshop*.

Klenk, M.; Molineaux, M.; and Aha, D. W. 2013. Goal-Driven Autonomy for Responding to Unexpected Events in Strategy Simulations. *Computational Intelligence*.

Lelis, L. H. S. 2017. Stratified Strategy Selection for Unit Control in Real-Time Strategy Games. In *Proceedings of IJCAI-17*.

Mern, J.; Yildiz, A.; Sunberg, Z.; Mukerji, T.; and Kochenderfer, M. J. 2020. Bayesian Optimized Monte Carlo Planning. *CoRR*.

Molineaux, M.; Klenk, M.; and Aha, D. 2010. Goal-Driven Autonomy in a Navy Strategy Simulation. In *Proceedings of AAAI-10*.

Paredes, A.; and Ruml, W. 2017. Goal Reasoning as Multilevel Planning. In *Proceedings of the ICAPS-17 IntEx Workshop*.

Silver, D.; and Veness, J. 2010. Monte-Carlo Planning in Large POMDPs. In *Proceedings of NIPS-10*.

Yoon, S.; Fern, A.; Givan, R.; and Kambhampati, S. 2008. Probabilistic planning via determinization in hindsight. In *Proceedings of AAAI-08*.

Zinkevich, M.; Johanson, M.; Bowling, M.; and Piccione, C. 2007. Regret Minimization in Games with Incomplete Information. In *Proceedings of NIPS-07*.