Abstract

Pandemic™ is a popular cooperative nondeterministic board game with a state space and game tree that are far too large to represent fully. In this project I cast the game as a stochastic planning problem to be played by a single agent. I create Monte Carlo Tree-Search agents to play the game, and specifically utilize the canonical "Upper Confidence for Trees" (UCT) method. I attempt to improve UCT agents by replacing the canonical win/loss reward with state heuristic valuations. I then test new search agents that replace the reward gleaned from random games entirely with the output of state value heuristic functions ("heuristic-guided UCT"). In both settings I use different human-constructed state value heuristics to find which encoded information might be most important to agent performance. I also test the efficacy of a selection policy that greedily chooses the highest Expectimax-valued action.

The first finding is that agents using Expectimax selection perform much better than those using a canonical UCB1 selection policy. Results also clearly indicate that random games make for a poor estimate of state value regardless of the modifications used. Replacing the reward of a random game with the output of a hand-crafted state valuation function, while maintaining the same UCT search framework, results in dramatically stronger play. Performance trends indicate, though, that achieving human-level performance with these agents still may not be computationally practical.

Contents

1 Introduction

1.1 Purpose

The purpose of this project is to learn about Monte-Carlo tree search (MCTS). My plan was to learn by building and testing such agents in an interesting game of my choice and construction. To this end, I gave myself the primary goal of building a working MCTS agent that implemented the canonical methodology - Upper Confidence for Trees (UCT) - in its most simple form. Thereafter, I would find the degree to which human-encoded knowledge could be used to improve it. Based on experimental results, I could try modifying the methodology to further improve performance. This incremental approach to learning obviously comes at the cost of not allowing for exposure to the most recent applications of MCTS, but does come with the benefit of providing a direct and substantial amount of experience with the underlying method.

1.2 Central Questions

My purpose inspires a few key questions. They are primarily directed at being able to characterize the ability of agents to play the game and the reasons *why* their performance is what it is.

1. How well do UCT agents perform in the board game *Pandemic*?
2. To what degree can human-encoded heuristics improve the performance of these agents?
3. What agent parameters and behavior correspond to the best and worst performance?

1.3 Structure

This report is structured into four primary sections: Background, Implementation & Experimental Design, Results, and Conclusion.

In the **Background** section I present the building blocks for casting *Pandemic* as a stochastic planning problem. This requires a discussion of Markov Decision Processes and classical planning. I also try to illustrate the core theory behind MCTS and explain the heuristic-guided UCT agent within the same context. I show the modifications to MCTS that must be made for a stochastic game. Lastly I explain the rules of the boardgame *Pandemic* and describe its complexity concerns.

In the **Implementation & Experimental Design** section I explain how the game, agents, and experimental environment were built. I begin by describing my implementation choices for the game itself, which requires a thorough description of some key methods and some explicit definitions of terms used. Then I describe how the agents used for experimentation were designed and implemented, together with the algorithms that represent their decision making process. In the final part of this section I describe the form of experiments in the project and provide a full list of all of the agents tested.

In the **Results** section I provide a sequential discussion of agent performance, beginning with that of a random agent that represents the default policy of my UCT agents. I also describe a human benchmark and use these two sets of played games to describe a shared set of attributes with which to measure agent performance. Then I characterize the UCT and heuristic-guided UCT agents by their ability to find strong lines of play as measured by different attributes of their constructed partial game trees and actions taken.

Next I simply describe the performance of each such agent according to the pre-

scribed game attributes. This includes a comparison of a semi-random heavy-rollout UCT agent with an equivalently powered heuristic-guided UCT agent, and a brief exploration of the effect of simulation budget on overall performance.

Lastly I try to dissect the behavior of agents by looking at strategic and tactical considerations that are or are not made by different agents. This directly informs conclusions about strengths and weaknesses of these agents.

The last section is the **Conclusion**, where I summarize my findings and indicate directions that I believe future work could take. I conclude my report with a brief reflection on what I've taken away from the project.

2 Background

2.1 Markov Decision Processes

Markov Decision Processes (MDPs) are a theoretical description of a dynamic environment typically used in the context of games. Their description is useful for the description of games because it can encapsulate the notions that (a) players have a choice of actions available to them during a game, which will change the game state, (b) the results of these actions may not be certain, and (c) each player gets some total reward based on the sequence of actions they choose. There are different formalizations of MDPs available. Here, I will introduce MDPs as a 6-tuple

$$\mathcal{M} = (S, s_0, A_s, P_a, R_a, L)$$

where:

- S is a set of states.
- $s_0 \in S$ is an initial state.
- A_s is the set of actions available in a given state s.
- P_a is a function $S \to \Delta S$ mapping a state $s \in S$ to a discrete probability distribution over successors in S under choice of action a.
- R_a is a function $S \times S \to \mathbb{R}$ mapping a transition $s \to s'$ to a real-valued reward under choice of action a.
- L is a function $S \to 2^{\mathrm{AP}}$ assigning each state $s \in S$ to a set of atomic propositions (AP) called *labels*.

A very small example MDP for a game with three states is shown in figure 1. This MDP has three states s_0, s_1, and s_2. There are three actions (labeled as 1, 2, and 3, shown as solid circles) available in initial state s_0, some of which may result in the MDP

transitioning in-place to state s_0. Only action 2 has a chance of reaching the state s_1 labeled as a winning one ("W").

Typically MDPs will be used in the context of games in order to evaluate and choose among potential strategies. A *policy* π encapsulates the formal notion of a strategy as $\pi(s) : S \to A_s$, which makes it responsible for choosing an action in every state. Sometimes (though virtually never in the case of complex games), an MDP can be completely specified and an optimal or close-to-optimal policy can be computed using methods like *Value iteration* (Bellman 1957) and *Policy iteration* (Howard 1960), both of which were described very early in the field of computer science. These methods require a full specification of the system, but come with guarantees about the optimality of the resulting policy.

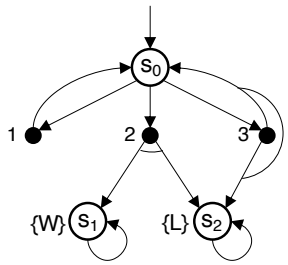


Figure 1: Toy MDP example. Explanation in main text.

Both methods rely on estimating a *value* $V(s) \in \mathbb{R}$ of states, which in the context of a game represents the ability of a player (or players) to perform well and/or win. There's a number of ways to use both methods for different games, so here I just try to describe the most canonical methods.

In the case of value iteration, state values are first initialized according to whether or not they're winning ($V(s) = 1$), losing ($V(s) = 0$), or neither (one can choose an arbitrary $c \in [0, 1]$ to initialize $V(s) = c$). Then they're updated iteratively according to the values of potential successor states and whether or not they want to be minimizing

or maximizing value. This is done by using the Bellman Equation[1] (below).

$$V(s) := \max_a \mathbb{E}_{P_a} \left[R_a(s, s') + \gamma V(s') \right]$$

The factor γ is the *discount factor* and lies in $[0, 1]$. Discount factor values close to 1 force $V(s)$ to incorporate state value and action rewards from distant parts of the MDP, while values close to 0 make more immediate rewards much more heavily weighted. After having found the values of states within some tolerance, one can choose an action that maximizes or minimizes the expectation of successor state value.

States that are far away from terminal states in the MDP will require many iterations of this process to see their initialized value begin to change. This means that the required number of iterations for convergence is typically prohibitively large for complex MDPs. Further, the complexity of the calculation grows with the size of the state space and the connectedness of the MDP. Both of these considerations make it an impractical approach for complicated games.

Policy iteration starts by defining a policy with random assignments $a \in A_s$ for each $s \in S$. Then two processes are computed iteratively: (1) The value of each state is updated according to its value under the current policy, and (2) The policy is altered at each state to choose an action with the highest value. An optimal policy is the fixed point of this process. Like with value iteration, its complexity is tied to the size of the state space and makes for an impractical method in complex games.

2.1.1 Markov Decision Processes for Stochastic Planning

Planning is a field of study in Computer Science in which problems are represented by a *planner* being tasked with taking a sequence of *actions* that achieve a *goal*. The goal is comprised of a logical composition of requirements that must be met in order for a

[1] A perhaps more familiar formulation has reward function $R : S \times A \to \mathbb{R}$ denoted by $R(s, a)$ as a fixed quantity outside of the expectation. The formulation here assumes that rewards are the result of the observed transition rather than choice of action.

planner to have succeeded. Often, *how* the goal is achieved is immaterial to a solution. *Stochastic* planning problems are planning problems in which actions are stochastic and may result in an outcome drawn from a known probabilistic distribution.

One example of a planning problem is a scheduling problem: given long lists of students, teachers, and classrooms, generate a schedule where no teacher has too many students at a time, students have at least five classes a day, etc. A schedule planner could cast the problem as one in which they have to assign students and teachers to classrooms one-by-one in sequence, and return the schedule that results from appropriate solutions.

An important property of planning problems as opposed to other closely related problems is that typically there's no notion of *cost* associated to a solution sequence. When the schedule planner above is assigning students to classrooms as part of trying out a solution, it doesn't cost them anything to make the assignment. Further, classical planning holds that *any* solution is just as good as another: one satisfying schedule from a scheduling agent is just as good as any other, as long as it meets the requirements. Introducing any preference to solutions turns the problem into an *optimization* problem. Often optimization and planning problems are closely related, since the set of admissible solutions to any optimization problem could be considered as a planning goal over the parameters to be optimized.

Because no action has any inherent value in a planning problem and all solutions are equally valid, an MDP for a game that corresponds to a stochastic planning problem has that $R_a = 0 \ \forall a \in \bigcup_{s \in S} A_s$. This means that the value of any given state $V(s)$ is determined solely by the value of potential successor states.

While R_a could be made nonzero for "useful" actions in order to help inform agents, they are theoretically unnecessary. Their use in an MDP would correspond to a game that isn't strictly a planning problem, and so adds a qualification to the representation. This qualification would apply to derived policies that take advantage of nonzero re-

wards and leaves open the possibility that are not expert planners in the original context.

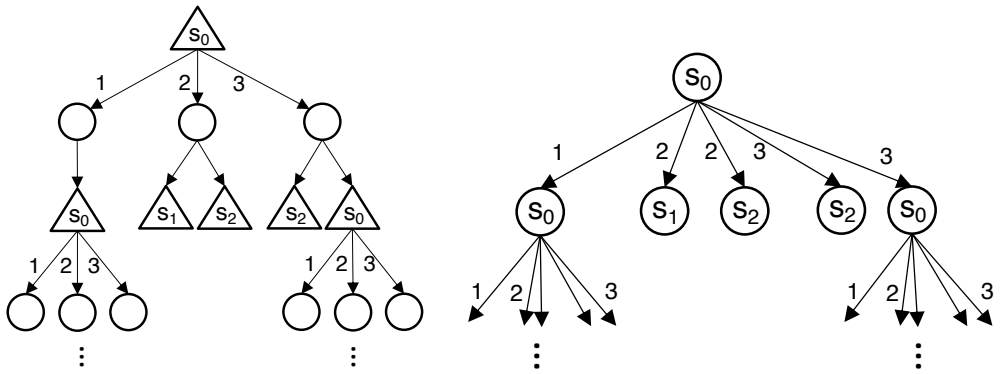
Now I want to focus specifically on planning as it applies to games. Single-player finite games with no notion of "final score" can be cast as planning problems, since the player only cares about being able to win and there's no reason to prefer any one method of winning over another. These games can be cast as an MDP with a subset of states labeled $\mathcal{W}$ (winning) and $\mathcal{L}$ (losing) in which the player wants to maximize their chance of reaching winning states from the current game state. The best possible player corresponds to an optimal policy $\pi^\star$ who can traverse the corresponding MDP along a path that maximizes $V(s)$ =$\Pr_\pi$(player eventually wins from s). The only thing that strictly matters is that $V(s) = 1$ for winning states and $V(s) = 0$ for losing states. If considering methods with which to calculate $V(s)$ then nominally we'd want that $\gamma = 1$, since all wins should be considered equal in the eyes of a planning policy π, regardless of their distance from the current state.

2.2 Stochastic Game Trees, Expectimax, and Tree-Search Planners

2.2.1 Stochastic Game Trees

A *Game Tree* is a theoretical construct that captures all possible game histories from an initial state s_0 of the underlying game MDP. It is formed by a directed acyclic graph (DAG) whose paths correspond to game histories. Figure 2a shows part of a game tree constructed for the MDP in figure 1. The chance elements of the game are represented by circles called *chance nodes* of the game tree, while the triangles correspond to MDP states from which an agent is responsible for decision making. These are called *decision nodes*. It is important to notice that even though the original MDP was incredibly small, the corresponding game tree is infinite. This is precisely because actions 1 and 3 can result in a return to the initial state s_0, and so the set of all possible game histories has no end. The nodes of a game tree do not inherently track whether or not a state

has been visited before.



(a) Game Tree (b) Search Tree

Figure 2: Turning the MDP in figure 1 into a game and search tree. Explanation in main text.

2.2.2 Expectimax

State values $V(s)$ in an MDP could be explicitly determined with a full specification of the states and an application of value or policy iteration. In a single player stochastic game with winning states valued at $V(s) = 1$ and losing states at $V(s) = 0$, this value corresponds to the maximum achievable probability of winning from any given state given an optimal policy $\pi^\star$.

The Expectimax algorithm (shown in algorithm 1) takes a node in a finite game tree and returns the maximum expected reward starting from the corresponding state of the game. The Expectimax value of a node, then, is exactly equal to $V_{\pi^\star}(s)$ for stochastic planning games.

Algorithm 1: Expectimax Algorithm

Function Expectimax*(Tree Node* n*)*:
  if $n.STATE$ is terminal **then**
    return V(n.STATE) `/* 1 for` s `winning, 0 losing */`
  if n is a decision node **then**
    return $\arg\max_c$ {Expectimax(c) | $c \in$ n.CHILDREN}
  else
    return $\sum_{c \in \text{n.CHILDREN}} \Pr(c) * \text{Expectimax}(c)$

For game trees that correspond to real-world games, an explicit calculation of Expectimax over the full game tree is impossible. The central limitation is the complexity dependence on the size of the game tree, which can be vastly larger than the (perhaps already intractable) size of the state space.

2.2.3 Tree Search Planners

Tree Search is a method for finding a path from an initial state to a pre-specified set of goal states. A tree formed by a DAG is constructed to represent the successive application of actions taken in an effort to reach the goal, and then a best action is chosen based on the construction. Some instantiations of tree search use the notion of *path cost* in order to be able to discriminate between solutions and choose an optimal one. Figure 2b shows a search tree derived for the MDP in figure 1. One can see that the search tree, even though it doesn't make chance nodes explicit, incorporates the notion of chance by showing that with each application of an action to state s_0 the result may be different. This makes it equivalent to the game tree: every *decision node* of the game tree corresponds to exactly one node of the search tree.

One important distinction between game trees and search trees is that game trees are a theoretical construct that might even be specified in the problem, but a search tree is a *tool* that has to be constructed to represent part of the potentially infinite game tree. What makes search trees useful is that as long as an entity (called the *generative model* for the game MDP) can tell the tree-constructor what states come from

applying each action, we don't have to know what the full game tree is. We can make a partial game tree of potential short-term game histories and make a choice of action(s) based on that. This is to say that one doesn't *make* game trees to solve complex games, but one can *approximate* it with a search tree in order to inform decision making.

A Tree-Search *planner* is a kind of planning agent that uses a partial construction of the game tree in order to choose a sequence of actions that will move it towards achieving the planning goal. The previous example of a scheduling agent that assigns timeslots to students and teachers one-by-one in order to find a valid schedule is an example of such a planner.

2.3 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a fairly recent and popular kind of tree search agent. Its popularity comes from its consistently powerful application to otherwise difficult games. Whereas classical informed-search methods like A* rely on *exploiting* paths that appear most promising until proven otherwise, MCTS capitalizes on the idea of interleaving the *exploitation* of promising paths with the *exploration* of others. Further, MCTS methods have shown a lot of aptitude for adaptations with offline-learned models (though those will not be explored here).

The core of MCTS theory comes from a statistical problem called the *Multi-Armed Bandit* introduced and explored formally in the 1950's. The problem was incrementally explored and adopted in the field of Artificial Intelligence and game-playing until in 2006 the first papers appeared that described its canonical form. In this section I'll walk through the background of MCTS to prepare for the treatment in this project.

2.3.1 The Multi-Armed Bandit Problem

The *Multi-Armed Bandit* problem describes a broad collection of scenarios, both real and theoretical, in which a finite amount of resources must be allotted among several candidate choices in order to maximize the total reward gathered from all of them. The amount and distribution of reward for each choice is mostly unknown, and typically the allocation of the resource can be used to find out more about each distribution. The problem is very old and saw research in the first half of the twentieth century, though it was only studied in separate applications.

In the canonical model of the problem, there are K unknown reward distributions $\Delta_1, \Delta_2, ..., \Delta_K$ and N total samples that can be drawn across them all. The problem is to find an allotment strategy for the N samples that will maximize the expected sum of sampled rewards. Importantly, this formalization requires that a successful allotment optimize the balance between the *exploration* of less-tested distributions and *exploitation* of high-valued distributions.

An important quantity to study in the context of these problems is *regret* ρ, which can defined using $\mu^\star$ to denote $\max(\mathbb{E}(\Delta_i))$ and r_i to denote the reward gathered from sample i (eqn. 1), or alternatively with $T_i(N)$ to denote the expected number of times that choice i will be taken in N total samples (eqn. 2):

$$\rho = N\mu^\star - \sum_{k=1}^{N} r_i \tag{1}$$

$$\rho = N\mu^\star - \sum_{k=1}^{N} \mu_i T_i(N) \tag{2}$$

This quantity represents the difference between the maximum achievable reward expectation and that realized by the agent. Larger values indicate that the cumulative sampled rewards were much less than that in the best possible allotment of resources, while small values indicate a near-optimal allotment.

In 1985, Tze Leung Lai and Herbert Robbins (the latter having formalized the multi-armed bandit problem in 1952) found that for certain families of reward distributions and assumptions about the bounds of $\lim_{N\to\infty} T_i(N)$, the asymptotic regret could be bounded below by a multiple of $\log N$, and that this bound was the best-case regret in such cases (Lai and Robbins 1985). This finding established a standard by which selection policies could be judged: If regret grows by a multiple of $\log N$, then it is a good strategy. The specific selection policies that were suggested in the work were computationally intensive and required storing a history of sample rewards, which made them largely impractical to implement, though.

In 2002, Auer et al presented the UCB1 policy that built off of this and interceding work. They suggested a much simpler selection policy that maintained the same guarantees about asymptotic regret, and also provided a uniform convergence over parameter N rather than an asymptotic guarantee (Auer, Cesa-Bianchi, and Fischer 2002). The UCB1 policy seen in algorithm 2 shows the simple selection policy. The quantity $x_i + \sqrt{\frac{2\log(\sum_l n_l)}{n_i}}$ represents the average reward x_i drawn so far from distribution i together with an upper confidence bound that is designed to capture most of the probability mass above the existing average reward with high confidence. Like any good selection policy, this algorithm encourages both exploitation and exploration over the course of budget consumption. It was shown to abide by the best-case regret bound proposed in Lai and Robbins 1985.

Algorithm 2: UCB1 Policy

Input: Sample Budget $N > K$

foreach $1 \leq i \leq K$ **do**

  $x_i :=$ reward from one sample of i `/* Average sample reward */`

  $n_i := 1$ `/* Number of times sampled */`

while $\sum_i n_i < N - K$ **do**

  $\mathrm{k} = \arg\max_i x_i + \sqrt{\frac{2\log(\sum_l n_l)}{n_i}}$

  $r_k :=$ reward from making choice k

  $x_k = \frac{n_k * x_k + r_k}{n_k + 1}$

  $n_k = n_k + 1$

return $\arg\max_i x_i + \sqrt{\frac{2\log(\sum_l n_l)}{n_i}}$

2.3.2 Monte-Carlo Methods

Monte-Carlo methods are a kind of computational tool that relies on using random samples of a distribution or process in order to obtain some measurement of it. The applications of this methodology are vast, and are typically seen when some system is too large or complex to explicitly build and explore, but can be measured through the average behavior of random processes.

One could suppose, for example, that there's a remote computer that has been encoded with a program describing a normal distribution with some mean and variance, which can be queried for samples. It is our task to be able to find out what the mean and variance of that distribution are without having access to the source code. The most straightforward way to do this would be to generate a host of samples with repeated queries to the program, and generate estimates of the mean and variance. Because we're using the result of many *random samples* created by the process we want to measure, this is a Monte-Carlo method.

In the context of games, Monte-Carlo methods historically were used to estimate the value of states to inform a choice of action. In the most simple use, one could play N random games from the state (or states where nondeterministic) resulting from each of K actions, and choose the action that has the best average outcome (Abramson

1987). This approach could demonstrate a little success but did not come with theoretical guarantees outside of asymptotic convergence to optimality, and further was often outperformed in complex games by classical search methods informed by strong heuristics.

2.3.3 Monte-Carlo Tree Search

In 2006, two papers tried to apply Monte-Carlo and Tree-Search methods to the multi-armed bandit problem represented by designing a game policy π: what action should the policy choose when it has no inherent information about action value?

Rémi Coulom proposed an algorithm that applies the Monte-Carlo methodology to a tree search framework. In this algorithm, a partial game tree is instantiated only with a root node corresponding to the current game state. Then, the tree is iteratively expanded by playing random games from the root and adding only the first new game state to the search tree. When all of the successor states of a node are created, then the agent will use a non-random selection criterion to select an action rather than taking a random one. Statistics can be kept to choose better actions within the tree. This method draws a distinction between states represented by nodes completely internal to the tree (where action selection occurs with information), and those that are beyond the tree (where action selection occurs randomly). This separation helps to capture and use information gleaned from the rewards of random games traversed in different directions.

Kocsis and Szepesvári described an Upper Confidence for Trees (UCT) algorithm (Kocsis and Szepesvári 2006), in which a tree is instantiated with a root node corresponding to the current state. Then the tree is iteratively built by adding a new node representing the final action of a yet-untried action sequence, and collecting the result of a random ensuing game to update tree node values along that path. It relies on explicitly using a UCB1 selection policy at every completely internal node of the game tree.

The authors showed that this scheme resulted in a much more efficient and effective policy as compared to other Monte-Carlo based methods.

The name *Monte-Carlo Tree Search* (MCTS) was coined by Chaslot et al (Chaslot et al. 2008) to describe their modifications to the UCT method described above. They introduced the notion of a *heavy rollout*, wherein the default policy is modified by taking semi-random steps outside of the tree based on hill-climbing. In addition, this work provides a means to introduce expert knowledge to help inform the tree exploration component of search (see 2.3.6).

A popular survey of MCTS methods (Browne et al. 2012) explains that MCTS and UCT are almost interchangeable terms, with the qualification that MCTS is a wider umbrella that could be used to describe *any* online tree-search algorithm that incorporates Monte-Carlo methods.

In this project I'll use "MCTS" to mean precisely the UCT algorithm, though will use them interchangeably. I will also use a term "heuristic-guided UCT" to mean the UCT algorithm implemented with rewards given by a heuristic valuation function rather than a random game reward.

In UCT, a partial game tree is instantiated with a root node corresponding to the current state of the game. Thereafter, the agent will iteratively go through four steps.

1.) First, it will traverse the tree by starting at the root and selecting the highest UCB1-scored child until it's found a node that hasn't explored all of its children.
2.) Then it will add a new node onto the tree corresponding to the result of applying this unexplored action.
3.) Next it will play a random game from that resulting state.
4.) Lastly, each in the succession of actions selected on this iteration within the tree has its visit count incremented up by one, and its average reward is updated using the reward of the random game played on this iteration.

These steps are illustrated in figure 4, and the logic is shown in more detail in algorithm 3 (based on that from Browne et al. 2012). Classical UCT Search corresponds

to using (3a) on the third step with the terminal 1/0 game reward as a heuristic.

The policy that chooses actions beyond the partial game tree is called the *Default Policy*, and canonically is a random policy. Other applications of MCTS choose other default policies in order to collect better estimates of achievable reward. The *Tree Policy* determines which actions within the tree to explore on a given iteration, and canonically is a UCB1-selection at each level of the tree.

One can see in algorithm 3 that I exclude a description of BESTACTION (referenced in π_{UCT}). Typically, BESTCHILD is used in place of BESTACTION, resulting in a choice of action that maximized upper confidence bound for final selection. Other BESTACTION policies take into account average rewards, upper or lower confidence bounds, or even the number of visits.

Algorithm 3: UCT Search

Function π_{UCT} *(State* s_0*, Budget* B*)*:
- Instantiate root n_0 for state s_0
- **while** B not exhausted **do**
  - n_l = TreePolicy(n_0)
  - r = DefaultPolicy($n_l.STATE$)
  - BACKUP(n_l,r)
- **return** BESTACTION(n_0)

Function TreePolicy*(Node* n*)*:
- **while** n is nonterminal **do**
  - **if** n is not fully expanded **then**
    - **return** EXPAND(n)
  - **else**
    - n = BESTCHILD(n)
- **return** n

Function EXPAND*(Node* n*)*:
- a = Unselected Action from n.STATE
- Add new child n′ to n
- n′.STATE = Result of applying a to n.STATE
- n′.ACTION = a
- **return** n′

Function BESTCHILD*(Node* n*)*:
- `//` x_n `represents total observed average reward on all traversals divided by number of visits`
- **return** $\arg\max_{n' \in n.CHILDREN} x_{n'} + \sqrt{\frac{2\log(n.VISITS)}{n'.VISITS}}$

Function DEFAULTPOLICY*(State* s*)*:
- **while** s is non-terminal **do**
  - a = random action available from s
  - s = Result of applying a to s
- **return** V(s)

Function BACKUP*(Node* n*, Reward* r*)*:
- **while** n not null **do**
  - n.VISITS = n.VISITS + 1
  - n.TOTALREWARD = n.TOTALREWARD + r
  - n = n.PARENT

2.3.4 MCTS for Stochastic Games

Classically, MCTS was applied to two-player deterministic games. In this project I will be applying MCTS to a single-player stochastic game, which requires adaptation of the

underlying methodology to account for chance events.

The adaptation I use here is *determinization* of the problem. The method of determinization invents possible histories of "chance elements" of the game, which turns the chance events into deterministic events. An MCTS agent can use these generated histories to treat their planning as purely deterministic. Its power in reducing nondeterministic games to deterministic ones was famously used in the first master-level Bridge AI (Ginsberg 2001), and has become a staple in the application of AI methods, and especially MCTS, to nondeterministic games. Whenever one chooses to use more than one potential history of stochastic events in order to build the search tree, they will inevitably slow the growth in search tree depth. This is because each stochastic transition will have to be traversed multiple times in order to explore all of the determinized stochastic outcomes.

In a game that involves die rolls between player turns, for example, one could determinize the problem for an MCTS agent by pre-generating a few possible sequences of die rolls. On each iteration within π_{UCT}, the agent will randomly choose which of the sequences to use, and let the agent traverse the tree as normal, using that history to collect the result of any die roll events as necessary.

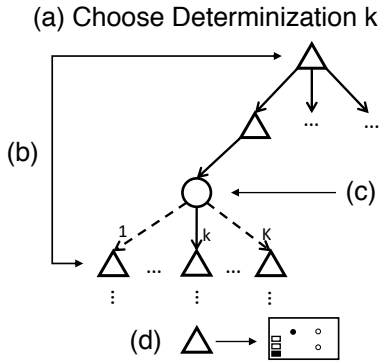


Figure 3: Illustration of a determinized search tree. Explanation in main text.

In some applications, each determinization is used to generate a completely separate search tree each of whose constructions can be done in parallel, and whose results can be conglomerated to make a final decision. Here, I use an approach that concentrates all the

learning into one tree by associating each simulation to one choice of determinization. I suspect that because this method is not as fast as parallelized methods, I am unable to find literature corresponding to this implementation. It does provide the benefit of not requiring a synthesis of action choices coming from multiple trees.

Figure 3 shows an illustration of the theoretical determinization method for the kind of search tree used in this project. After first selecting a determinization (a), deterministic nodes are traversed as normal (b). Chance nodes return the deterministic successor according to the chosen determinization (c). Lastly, the state from the generated node (d) can be returned for rollout.

In games with a lot of stochasticity (many possible chance outcomes or frequent chance events), this is incredibly useful because it replaces the complexity associated to the full stochasticity of the game with a sampling parameter K of user choice. Without determinization, search could be severely handicapped by the fact that almost every traversal of a stochastic transition in such games results in a state that wasn't seen before. Further, there would be no way to control whether or not some branches have gotten "lucky" or "unlucky" in their chance draws. By making K random determinizations at the beginning of search, one ends up with future game histories that (a) are shared across the search tree, and (b) still provide an unbiased estimate of the effect of the full stochasticity of the game. This choice of implementation induces a search tree that exactly corresponds to the one shown in figure 2b, with the exception that the number of deterministic successor states under any stochastic transition is always K regardless of the actual number of potential successor states.

This method requires that adaptations are made to the original UCT algorithm, which are shown in algorithm 4. They primarily affect the BESTCHILD selection, which now has to incorporate both (a) the fact that nodes may be chance nodes, and (b) that chance nodes require a choice of determinization. Otherwise, the only change

is that knowledge about which determinization is chosen must be passed down to the Tree Policy. The Tree Policy will also always return a deterministic choice node, even if a chance node was just created.

Algorithm 4: UCT Search modifications for using K Determinizations in Stochastic Games

```
Function π_UCT(State s_0, Budget B, Num. Determinizations K):
    Instantiate root n_0 for state s_0
    Make list of K determinizations
    while B not exhausted do
        D = Random determinization from list
        n_l = TreePolicy(n_0, D)
        r = DefaultPolicy(n_l.STATE)
        BACKUP(n_l, r)
    return BESTACTION(n_0)
Function TreePolicy(Node n,Determinization D):
    /* The tree policy still traverses through deterministic nodes.
       Must sometimes use determinization to resolve transitions */
    while n is nonterminal do
        if n is deterministic then
            if n is not fully expanded then
                return EXPAND(n)
            else
                n = BESTCHILD(n)
        else
            n = GETDETERMINISTICCHILD(n, D)
    return GETDETERMINISTICCHILD(n, D)
Function GETDETERMINISTICCHILD(Node n, Determinization D):
    if n is terminal or deterministic then
        return n
    if n has no deterministic child under D then
        Give n a deterministic child according to D
    return Deterministic child under D
```

2.3.5 Heuristic Modifications to UCT Search

In this section I describe modifications to classical MCTS that can be made using human-made *heuristics* that can help characterize better and worse game states in the absence of any other information. Here I use the term "heuristic" to refer to a real-

valued function whose domain is the state-space of the game. I constrain my heuristics to the range $[0, 1]$. Figure 4 shows the MCTS algorithm with modifications in (3a) and (3b). As mentioned earlier, classical UCT search uses method (3a) with only a win/loss (1/0) reward in the rolled out state. Lastly I'll mention how selection policies can be used to alter the final choice of action.

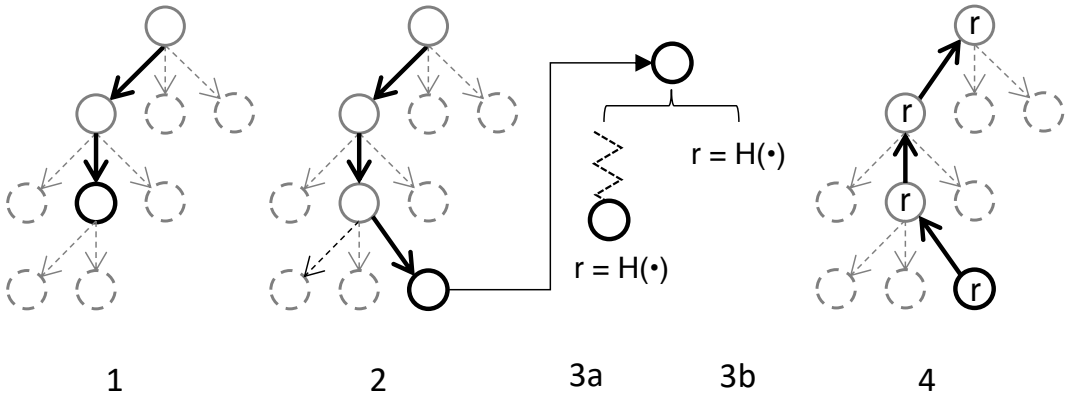


Figure 4: The MCTS algorithm with adaptation for the use of human heuristics.

One way to try incorporating heuristic knowledge into MCTS for stochastic planning problems is to replace win/loss rollout rewards with a handcrafted heuristic valuation on the resulting terminal state, as shown in (3a). The intuition is that in a planning problem, the result of (necessarily) poorly informed rollouts cannot expect to meet the planning goal and realize reward with any frequency. Whereas average rates of winning and losing might be a good estimate of value in two-player games, the rates of achieving a goal in a planning problem would be far too biased towards 0 to be of any use. One way to indicate the "goodness" of a position in the tree is to measure how close the rolled out state got to winning. States within the horizon of the game tree are *better* if the random rollouts from this state get closer to winning than those on other branches.

For this project I use heuristics that come from classical planning, which seemed directly applicable since the disjoint subgoals of the game are each the unique result

of an action with a fixed set of preconditions and in total comprise the entirety of the player objective ("planning goal"). The simplest heuristic measures what fraction of the subgoals for the planning goal are met ("Fraction Subgoals Satisfied"). A more complicated yet more informative heuristic can return what fraction of subgoals are satisfied *plus* what fraction of action preconditions are met towards achieving unsatisfied subgoals ("Fraction Subgoals + Preconditions"). I try using both of these heuristics in this project.

Another way to introduce heuristic knowledge is to replace the entire rollout with a heuristic evaluation of the state, as shown in (3b). This alteration actually changes the method from Monte-Carlo Tree Search to a heuristic-guided UCT search, since we're abandoning the Monte-Carlo method for value estimation. Unlike using the evaluation on a rolled out state which is almost certainly a losing one, one may want to incorporate knowledge of how close the leaf state is to *losing* as well as *winning*. In order to do so, one could make the same kinds of heuristics that evaluate how close game attributes are to inducing a loss ("Loss Proximity"). In addition, one could add more information by weighing the causal elements behind loss-inducing attributes rather than the attributes themselves ("Informed Loss Proximity"). I will attempt to use both measures of "loss proximity" in this project.

There's one last area for improvement to normal UCT Search: the means by which one selects actions upon exhausting the computational budget. I mentioned in an earlier section that in canonical UCT Search, an agent will select an action associated to the root child with the highest UCB1 score. There are many other ways to choose an action. One selection policy I will also test here is to take the highest Expectimax value child. In this selection scheme, I calculate the Expectimax value of each root child according to algorithm 1 on the partial game tree constructed by the agent. The one modification

that must be made is to incorporate the fact that when nodes aren't fully expanded, the algorithm returns the average backed up reward rather than the maximum of its existing children.

2.3.6 Similar work

In this section I want to describe some methods that also attempt to introduce human knowledge into the problem of finding good policies in complicated MDPs using search methods.

In 2001, Kearns et al. (Kearns, Mansour, and Ng 2001) described a method for generating approximately optimal online policies in a complex MDP by trading the state-space complexity of existing algorithms for complexity associated to the searched horizon depth H and stochastic sampling number C of a partial game tree. This method shares the idea of using "Sparse Sampling" of the stochasticity inherent to the MDP in order to estimate state values. Also, it relies on a generative model of the MDP rather than an explicit full or partial specification of the system. The algorithm takes the form of a step-limited (step maximum H) recursive calculation of action values $Q(s, a)$ with C stochastic samples per step. It relies on capturing the expected realizable rewards $R(s, a)$ within the horizon of the game tree, and by default does *not* include any heuristics in the valuation of depth-H states. They do provide a simple modification to incorporate state value estimation heuristics $\hat{V}(s)$ at step-terminal nodes, though.

I see two main differences between this method and the one described for this project. For one, this method does not rely on a fixed set of determinizations, but rather a unique re-sampling of C transitions at each state of the MDP. In addition, it is not any anytime algorithm and fails to provide solutions if it can't terminate.

The central benefit of the algorithm is its quantifiable optimality guarantees based on C and H and independent from the size of the state space. This guarantee came with

the central cost of complexity in $\mathcal{O}(C^H)$, which could still easily be untractable. As it pertains to the problem of stochastic planning, this algorithm would fail to perform since no action has any explicit reward. Further, even actions that have some seemingly implicit reward (e.g. those achieving a subgoal) are not common.

In 2008, Chaslot et al (Chaslot et al. 2008) introduce a way to explicitly incorporate human knowledge as part of exploration by modifying the original UCB quantity, called *progressive bias*[2]:

$$\frac{R(n)}{N(n)} + \sqrt{2\frac{\log N(n')}{N(n)}} + f(n, n')$$

This expression uses n' to represent the parent node of tree node n, $R(n)$ for total accumulated reward for node n, and $N(n)$ for visit count on node n. The function f is the heuristic, and originally was designed to decrease with $N(n)$ in order to guarantee convergence to true optimality and avoid any heuristic play pathology. In the rollout-replacement modification described above, I replace $\frac{R(n)}{N(n)}$ with the running average of traversed leaf-state valuations, so that the score held on each node is the average heuristic valuation of states in the ensuing subtree. In this context, then, we essentially use f as a replacement for realized rewards. This makes it possible to unintentionally introduce strong play pathologies.

In 2009, David Silver (Silver 2009) describes a large amount of learning methods as applied to MCTS. Offline-learned state heuristics could be used as part of search, as well as part of an informed default policy. Since the codebase for this project did not have any means to provide a flat vector representation of states let alone actions, I thought that using human-encoded heuristics on leaf states might be a cheap way to move in the direction of that method.

[2]This equation is not explicitly the same as that in the cited paper, but holds the same meaning

2.4 Previous Applications of MCTS to Stochastic Games

Since the introduction of MCTS it has been applied mostly to deterministic games, famously in Go (Silver et al. 2017), both with and without the application of offline learning. A few other recent applications have focused specifically on games with stochastic behavior.

Settlers of Catan is a popular nondeterministic and imperfect information board game approached by several of the same authors that had created MCTS (Szita, Chaslot, and Spronck 2010). Their brief investigation showed some ability against existing AI, but had a large reliance on computational budget for increasing performance. Modifications to the game may have also reduced the efficacy of the tested agents.

Mrs. Pacman is another nondeterministic game, with the additional requirement of near-realtime decision making (decisions must be made before the next frame is rendered). An MCTS implementation was proposed and developed (Gan, Bao, and Han 2011 and Pepels, Winands, and Lanctot 2014) that showed incredibly strong performance against all other agents. It specifically took advantage of the recycling of elements of the search tree between successive actions in order to economize on the small time budget.

The game *Hearthstone*™ is a digital two-player card game where each player uses cards from their hand in order to defeat their opponent. Because cards are randomly added to hands from a pre-existing deck, and the effects of some actions are randomized, much of the game is nondeterministic. Two recent works have explored the application of MCTS to this game.

In 2017, Santos et al (Santos, Santos, and Melo 2017) applied MCTS to the game. They modified UCT with the introduction of computationally intensive state value

heuristics during selection as in Chaslot et al. 2008, and used a kind of heavy rollout. These modifications greatly improved performance against current state-of-the-art AI agents by improving the win-rate from 21% (canonical UCT) to 42%.

In 2018, authors out of Warsaw applied MCTS with reinforcement learning modifications to the game (Swiechowski, Tajmajer, and Janusz 2018). They also implement determinizations to determinize the search, and explicity use the "cheating" (full knowledge of yet-to-be-uncovered events) in order to provide stronger games for offline learning much quicker. Unlike in the already described methods, they search over *Information Sets* by hashing the observable components of game states. This increases the efficacy of search by sharing rewards associated to states visited via multiple tree paths. Lastly, they also implement the methodology as in Silver 2009 in order to capitalize on offline knowledge to improve online search.

These agents exhibited completely dominant play over random agents and very strong play against an uninformed MCTS implementation. It could even outperform low-level players and win a few games against high-level players. In all the results are encouraging for an application to MCTS to stochastic games.

2.5 The *Pandemic*™ Board Game and Formalization

Pandemic™ is a board game released in 2008 by Z-Man Games. Its creation was directly inspired by the 2002-2004 SARS outbreak (Leacock 2020), and so was built to reflect the deeply uncertain and stochastic way in which disease spreads around the modern world. It is a nondeterministic cooperative game of complete information which always results in a win or loss for the players and is guaranteed to last a finite number of turns. There is no inherent notion of a "score" for the players, and so no way to discriminate between different wins and losses. These facts lead me to cast the game as a *stochastic planning* problem: It is the goal of a single agent to move the game into a winning state with a sequence of actions under uncertainty. Any winning traversal is just as valued

as any other, and any losing traversal is just as undesirable as any other.

In the remainder of this section I'll describe the rules of the original game (for a more a complete and thorough description of the game rules, one can refer to the rule manual[3]), as well as modifications that have been made for this project. Lastly, I will try to outline the complexity concerns of the game by estimating the size of the game tree and state space.

2.5.1 Original game rules

A picture of a *Pandemic*™ game board is shown in figure 5. The components labeled in the figure will be referenced in the ensuing discussion.

Pandemic™ is played cooperatively by 2-4 players on a discretized version of the world represented by a fully connected graph of 48 cities. Each player maintains a hand (item (g) in figure 5) of cards most of which can be traded with other players, used for special kinds of movement around the board, to build research stations, or most importantly to find cures for disease. Hands are supposed to be played openly, in full view of all players. Players can have at most 7 cards in their hand, which means that if any player is pushed over that limit at any time, they must discard down to the limit before the game can progress. Special cards called *Event Cards* (one of which is shown in player hand (g) in figure 5) can be held by players as part of their hand and used then discarded at any time, but not traded or used for any purpose but to implement their prescribed effect. Their effects are typically very powerful, for example being able to move any player to any position on the board, or preventing more cities from being infected at the end the current players turn. Each player is allotted a fixed *role* (item (b) in figure 5) at the beginning of the game that grants them either some special ability that they can use during their turn, or a special modification to the rules that applies

[3] https://images-cdn.zmangames.com/us-east-1/filer_public/25/12/251252dd-1338-4f78-b90d-afe073c72363/zm7101_pandemic_rules.pdf

only to them. On their turn, each player must use up to four actions to move around the board, trade cards, build research stations, treat disease, and/or find a cure.

Figure 5: Pandemic board set up with three players.

In addition to players, a game maintains two cards decks: a *Player deck* and an *Infect deck* (items (d) and (j) in figure 5, respectively). Whenever a player has finished using up to four actions on their turn, they have to draw two *player cards* from the *Player deck* in sequence. In the player deck there are 48 *City cards* (one for each city present on the board), 5 *Event Cards*, and 4-6 *Epidemic* Cards (number depending on chosen difficulty, with smaller amounts being easier). While the former two kinds of cards are put into a players hand when drawn, *Epidemic* cards (item (e) in figure 5)

are not, and instead will cause several things to occur. Firstly, the *infection rate*, which dictates how many cards are drawn from the *Infect deck* during that phase of a turn, moves to the next value in a fixed list of values. Then, the bottom card of the *Infect deck* is drawn and the city to which it corresponds is infected with 3 disease cubes of that color. Lastly, all of the *Infect cards* in the discard pile for that deck are shuffled and put back on top of the *Infect deck*. During the setup of the game, *Epidemic* cards are dispersed approximately evenly around the *Player deck* by segmenting the non-*Epidemic* cards into a number of equally-sized stacks equal to the number of *Epidemic* cards to be used, shuffling one *Epidemic* card into each stack, and then piling the stacks on top of one another.

After a player has drawn two cards from the *Player Deck*, they draw and resolve cards sequentially from the *Infect Deck* until they've drawn an amount equal to the *Infection Rate* (a number that starts at 2 and is at most 4 over the course of any game). The *Infect Deck* is comprised of 48 cards, each of which corresponds to exactly one city represented on the board. Whenever an *infect card* is drawn from the top of the *Infect deck*, the amount of disease on that city corresponding to its *color* is incremented up by one, and then the card is put into the discard pile for that deck.

Cities are segregated roughly by world region into four colors: blue, yellow, black, and red, which are the four types of disease present in the game. There are 12 cities of each disease color. Over the course of the game, plastic cubes of each color are placed on cities according to *infect cards* drawn at the end of each players turn. Disease cubes (item (f) in figure 5) track the amount of each disease present in a city. Unless a city is seeded with disease during the setup of the game, all cities start with zero cubes of each color. When a city already has three cubes of a color and the game would require placing a fourth, an *outbreak* occurs instead and a cube of that color is placed on all neighboring cities. This opens up the possibility of *chain outbreaks*, where multiple out-

breaks occur in sequence due to the attempt to place a single cube. The region shown by (h) in figure 5 is at risk of seeing a chain outbreak in the future.

Players immediately **lose** the game if any of the following conditions are met: there have been 8 or more outbreaks (tracked by counter (a) in figure 5) in the game so far, more than 24 disease cubes are required to track any disease, or a player was unable to draw from the *Player deck* at the end of their turn because it was out of cards. This last condition imposes a hard "time limit" on the game, and ensures that the full game tree is actually finite.

Players immediately **win** the game once they've found the cure to all four diseases, whose status is tracked by tokens in item (c) of figure 5. In order to cure a disease, a player must be at a research station (both of which are represented by tokens, as at (i) in figure 5), and have at least five cards whose color is the same as an uncured color. Performing this action removes those five cards from the players' hand.

2.5.2 Modifications in Implementation

For this project several changes have been made to the original rules of the game. Firstly, two *Event Cards* have been removed from the game: *Resilient Population* and *Forecast*. The former provides the ability to remove cards from the *Infect deck* discard pile permanently, therefore eliminating the chance that they'll ever be drawn from the *Infect deck* again. This is removed because unlike some other *Event Cards*, it doesn't need to be considered as an action during the player turn, since its effect on gameplay will only be realized when drawing an *Epidemic* card. In addition, it introduces a distinction between determinized game histories on different parts of the search tree, and so makes for very complicated implementation. I do believe that its inclusion would make for some interesting experimentation, since it allows the player to directly control

the stochasticity of the game.

Forecast allows the player to reorder the top 6 cards of the *Infect deck*, and so partly determinizes the near-term game horizon. The difficulty is that this choice requires a preference over 6! orderings of cards (since generally one has to assume that order matters, even if it typically doesn't). In such a case one could either hard-code rules for potential orderings, which would undercut any success an agent has with caveats about encoded human knowledge, or randomize the choice, which could potentially damage or help the agent just like existing stochasticity and so add nothing to the problem at hand.

Further, the *Dispatcher* and *Contingency Planner* roles have not been implemented in the project. The former role has the ability to move *any* player using regular movement options available considering the *Dispatcher*'s hand, and so substantially increases the complexity of decision making while also being difficult to implement. The *Contingency Planner* is allowed to use an action on their turn to reclaim used *Event Cards* from the discard of the *Player deck* without having them count towards their hand-size constraint. Because the number of *Event Cards* have already been reduced in implementation, and the use of this special ability is already constrained by the presence and use of *Event Cards* to begin with, I decided that the potential utility derived from this role would be too low for exploration.

In addition to the removal of some elements of the game, others have simply been modified. In the original rules, *Event Cards* can be used at any time during the game, as long as it is not during the resolution of a player action or card draw. Here, these cards may not be used outside of the player turn. Two *Event Cards* can only be used at the beginning of the player turn (or upon a discard being forced on the player): *Government Grant* and *Quiet Night*. This choice leverages the fact that if it would be a good option to use during the player turn at all, then it might as well be used at

the beginning of the player turn. This provides a modest search space optimization. *Airlift* has been made available to players only (a) at the beginning of their turn, or (b) after having built a station, cured or treated disease, or traded cards. This drastically reduces the impact that *Airlift* has on the branching factor seen by search agents, and implicitly prevents obviously poor lines of play that include expending actions moving and *then* using *Airlift*, even if such prevention might cost the performance of very weak agents by preventing course correction. By virtue of the fact that cards can't be played outside of the player turn, *Event Cards* have been reduced in power, and so introduces a minor handicap on agent performance.

The legality of some actions has been restricted to remove dominated (in the game-theoretical sense) actions from the search space. I've made a few such modifications. For one, while the original rules make it nominally legal to discard a card to travel to a neighboring city (or even the city a player is currently in), my implementation would call such moves illegal. Additionally, a player may not choose to do nothing and expend an action when it could instead treat disease or find a cure. In both of these modifications, I've eliminated strictly dominated actions that could never be the "only good choice", even if they end up being just as good as others.

Some other actions have been removed from the search space in an effort to reduce complexity, though unlike above *without* the guarantee that they're dominated. In general, an agent can't move to their *last* position unless they've treated or cured a disease, built a station, traded cards, or haven't moved yet this turn. This does *not* represent the elimination of strictly dominated actions from a given state, since it is entirely possible that movement away from the last position was suboptimal, but moving back would be optimal. In addition, the *Airlift* card precludes its use to move a player to a neighboring city, which is a very small but almost always sensible optimization.

Lastly, while the *Discover Cure* action allows the player to choose which cards of the to-be-cured-color to discard, this implementation will blindly remove cards of that color until the required number of cards has been met.

2.5.3 *Pandemic*™ as a single player game

Pandemic™ is a cooperative game, and so can be cast as a single-player game by introducing an agent that's responsible for making a choice of player action during every player turn. I have chosen in this implementation to cast the game as such since (1) all players share the common goal of winning the game, (2) there's no notion of individual reward and so players can never disagree on priority, and (3) the game rules make explicit that information is supposed to be shared openly between players.

2.5.4 Deterministic Branching

The minimum number of actions that could possibly be under consideration by the agent is 2: one option to do nothing, and one option to move to a single neighbor. This requires that a player have no cards, that there are no event cards present, that there's no disease present in their city, and that they're at the one city in the game that has one neighbor.

Typically, players will have the option to move to any of at least 3 neighbors, the option to do nothing, and the option to discard any city card in their hand to fly to other parts of the world, all of which adds up to a typical consideration of 6-10 actions at each step.

At most, an agent could be presented with a choice over about 250 actions, which comprises a choice over potential uses of all *Event Cards*, movement to neighbors, trading cards with players in their city, moving from their current city to virtually any other on the board, and of course doing nothing.

2.5.5 Stochastic Branching

Because the game incorporates frequent elements of chance, it is also a *Stochastic* single player game. Typically, game play will consist of an agent making four deterministic actions (barring the use of event cards, which would increase this number but not expend an action) before arriving at a chance node, which will then bring the game to a random deterministic successor. In general the number of possible stochastic transitions is the number of sequences of two *player card* draws followed by at least two *infect card* draws.

The minimum stochastic branching factor might be 4[4], which would only be the case when there are two remaining *player cards* (and so two sequences from which to draw them), and two known *infect cards* on top of the *Infect deck* (similarly two possible sequences). This could only be achieved near the very end of the game, and so never more than once in a game if it occurred at all.

More typically, there may be about 20-30 player cards left in the player deck (including *Epidemic* cards), and about 5-10 *infect cards* on top of the *Infect deck* from the last *Epidemic*. This would put the number of card-draw sequences, and therefore stochastic branching factor, at about $\mathcal{O}(10^5)$, which doesn't account for the fact that the draw of an *Epidemic* card introduces the draw of a card from the bottom of the *Infect deck*. Even if you tried to greatly reduce the branching factor by the collapsing of equivalent sequences (sequences that result in exactly the same final state), you'd be able to reduce the size of branching by a factor of 2 to 6 (depending on *infection rate*), which is a relatively modest reduction.

At most, and pretty much always at the beginning of the game, the stochastic branching factor is $\mathcal{O}(10^6)$, which incorporates the fact that the *Player deck* is largest at this point, and that draws from the *Infect deck* are coming randomly from among all unseeded cities.

[4]There is a contrived case wherein one player card remains undrawn (stochastic branching factor of 1), which will lead to an immediate loss upon the attempt to draw the second, but I do not consider this case.

2.5.6 Game Depth

Here I consider depth in both the number of decisions that an agent must make in a game and the number of stochasticities that must be traversed (i.e. complete sequences of card draws following a player turn).

The minimum total game depth is 5, which corresponds to the exceedingly rare but possible event that a player makes 4 choices (4 decisions), and then the game is immediately lost when the first cards of the game are drawn from the two decks (1 stochastic transition).

In my experience playing, players might win or lose when there are about 10 *player cards* left, which in the case of a 3-player game would correspond to at least 18 player turns, each of which is 4 actions, yielding $18 * 4 = 72$ player decisions. On top of that, you'd have a stochastic transition after each of 18 player turns, resulting in a total depth of $72 + 18 = 90$.

At most, players might not lose or win *until* the *Player deck* is out of cards. Further, they could have tried to hoard all the cards they've ever gotten, forcing them to make a choice of discard every time they draw a new card at the end of their turn, which would begin happening after a few player turns. In the case of a 3-player game, this results in over 130 decisions being made (4 action choices on each turn and 2 additional discard choices after almost every turn, for 23 turns), and $23 * 3$ stochastic transitions (one before each deterministic discard action is made, and one after), resulting in a total game depth of 200.

2.5.7 Game Tree Size

Here I put together all of the information provided so far to try to characterize the size of a game tree that represents games playable by an agent.

These values allow us to estimate an upper bound on the number of playable games at $(250)^{130}(10^6)^{69} \approx \mathcal{O}(10^{724})$ and so safely qualifies as intractable for classical AI search

Tree Property	Est. Average Value	Est. Highest Value
Deterministic Branching	6	250
Deterministic Depth	72	130
Stochastic Branching	10^5	10^6
Stochastic Depth	18	69

methods.

A more grounded estimate of the size of the game tree incorporates estimates of average values traversed in the tree: $(6)^{72} * (10^5)^{18} \approx \mathcal{O}(10^{146})$, which is much smaller value but whose order of magnitude is still an order of magnitude away from approaching tractability for classical search methods.

2.5.8 State Space complexity

Here I will try to estimate an upper bound on the size of the state space for *Pandemic*. I'll first try to establish an upper bound with a naïve estimation of the bounds on individual attributes. I then refine this estimation by trying to account for the facts that some complexity measures are causally correlated, and that much of the state space is virtually impossible to explore for any given game.

Without making any simplifying assumptions or groupings of states, states are represented purely by the collection of logical and numerical attributes that entirely defines the configuration of the game. Once a game is set up, a few attributes (like what *role* each player is assigned) might be fixed, but others are subject to change during the course of the game. Below I've enumerated those attributes and tried to either calculate or estimate an upper bound for the number of legal values that each might take, regardless of whether or not they're all reachable during any game.

- The position of N players $= 48^N < \mathcal{O}(10^7)$

• The content of N players' hands	$< (\sum_{i=0}^{7} \binom{51}{i})^N \approx \mathcal{O}(10^{32})$
• The cards that remain in the *Player deck*	$< \sum_{i=1}^{51-N*2} \binom{51}{i} \approx \mathcal{O}(10^{12})$
• Configurations of the *Infect deck*	$< \sum_{i=1}^{48} \binom{48}{i} \approx \mathcal{O}(10^{14})$
• The amount of each type of disease on each city	$< 4^{48} \approx \mathcal{O}(10^{28})$
• The position of research stations	$= \sum_{i=1}^{6} \binom{48-(i-1)}{i} \approx \mathcal{O}(10^{9})$
• Whether or not each disease is cured	$= 16 \approx \mathcal{O}(10^{1})$
• Whether or not each disease is eradicated	$< 16 \approx \mathcal{O}(10^{1})$
• How many outbreaks have occurred	$= 8 \approx \mathcal{O}(10^{1})$
• What the *infection rate* is	$= 3 \approx \mathcal{O}(10^{0})$
Naive upper bound:	$\approx \mathcal{O}(10^{105})$

In order to do try calculating an upper bound on the size of the state space, we can just try adding the order-notation exponents on each attribute, yielding an upper bound on state space of about $\mathcal{O}(10^{105})$. I would expect this to be a vast overestimate of the true size of the state space, and an even more drastic overestimate of the number of traversable states in any given game.

Firstly, the complexity of the player deck and player hands are related - a better estimate would try tracking the number of combinations of player hands and the number of permutations for each corresponding player deck. The result might be something like $\mathcal{O}(10^{33})$ total between the deck and player hands[5]. The complexity associated to the arrangement of the *Infect deck* is also mostly unrealizable since *Epidemic* cards will keep putting discarded cards back on top, effectively reducing the complexity to perhaps $\mathcal{O}(10^{6})$. This estimate is based on the maximum amount of discarded cards that could be put on top of the *Infect deck* (~ 20). In addition, the initial setup of the game is going to typically dictate that the gameplay will traverse through a relatively small subset of city infection permutations, maybe reducing that complexity to $\mathcal{O}(10^{15})$ by assuming only 24 cities get infected during each game . I would also expect that

[5]This is the result of an estimate for 4 players: (# of permutations of 51 - 4*7 cards)*(51 choose 7)*((51-7) choose 7)...

in most games, traversable research station complexity won't be close to $\mathcal{O}(10^9)$, and might estimate it closer to $\mathcal{O}(10^5)$[6]. Lastly, eradication status is tied to cure status and disease presence, and so even if there's a few legal configurations it does not in of itself contribute to complexity.

All of this reduction leads me to a more sensibly derived upper bound on the size of traversable state space: $\mathcal{O}(10^{69})$. Even though this is an upper bound, its magnitude presents a seriously intractable challenge for classical AI methods. Further, state-lumping approaches would require state equivalence classes to group together gigantic sets of states in order to make it meaningfully less complex.

[6]For games with an *Operations Expert* I would expect the original complexity estimate for traversable configurations of reserach stations to hold. My experiments don't include this *role*, so I aim to achieve a lower estimate here.

3 Implementation and Experimental Design

The goal of this section is to provide a description of the high-level methods of the game, agents, and experiments. I will provide the algorithms that correspond to implemented methods, as well as more high-level descriptions to capture relationships and dependencies.

All of the code that captures game, agent, and experimental behavior is written in C++ compiled to the C++17 standard. It was developed, tested, and debugged from scratch over the course of May through early July of 2020. Some more detailed implementation details can be found in the appendix.

This implementation represents a significant project contribution in the form of a relatively unique new game environment that hosts a variety of functionality outside of providing a generative game model. It required a substantial amount of time for design, coding, and testing. Previous implementations I found were focused around providing for an easy GUI representation, were handicapped by substantial modifications to the game rules, or were not in a language amenable to very fast game traversal.

3.1 Game Implementation

The high-level methods described here rely on a `Board` object that represents the game state, which holds all of the information about the players, the position and amount of disease present, the status of both decks, and more.

3.1.1 Game Logic

Whereas the `Board` represents the *state* of the game, the `GameLogic` provides a generative representation of the underlying MDP by providing for possible *Actions* as well as information about the status of the current `Board`. Figure 6 shows a high-level depiction of the game logic attributes that drive gameplay. Item (a) is an `active_board` which is a `Board` object that is kept by the game logic to represent to the current state of the

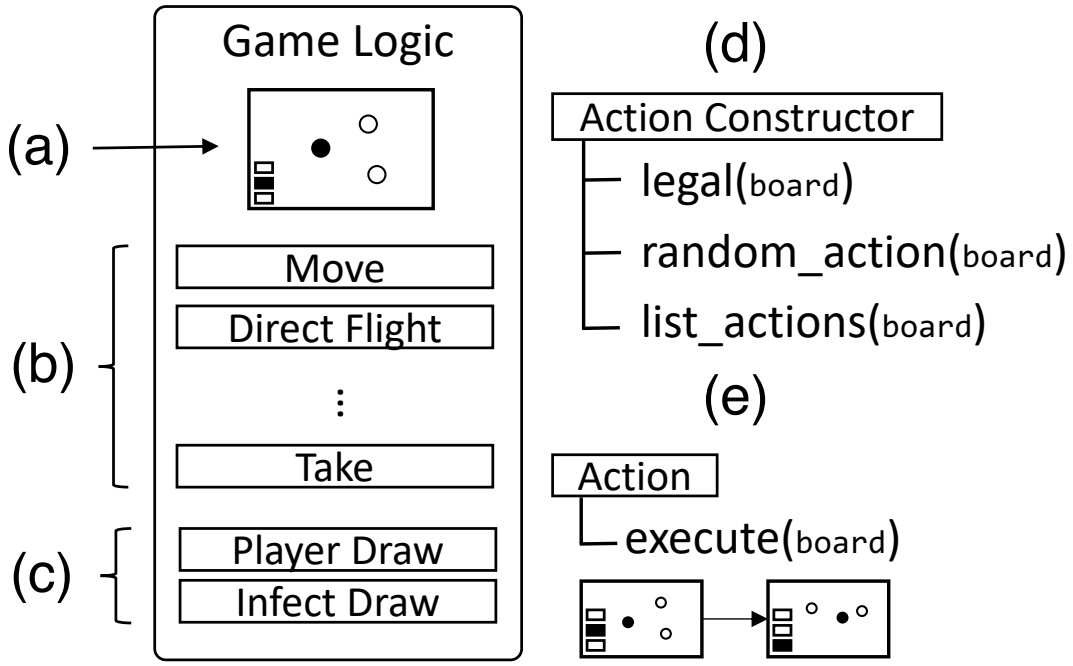


Figure 6: A high-level illustration of game logic attributes

game.

Items (b) and (c) both point to the `ActionConstructor` (item (d)) attributes of the game logic. These entities are responsible for taking any `Board` object and being able to:

- Tell whether or not *any* `Action` to which they correspond is legal
- Generate a random legal `Action`
- Generate a list of all legal `Actions`

`Actions` (item (e)) generated by these `ActionConstructors` are a very simple class with a string representation (`repr()`, not shown) and an `execute(Board)` method that performs the prescribed effect on the given `Board`, and so advances the state. Some `Actions` have attributes that represent arguments, while others don't. Both `Actions` and `ActionConstructors` correspond to *types* of actions like movement between cities,

curing disease, or trading cards.

Item (b) points to the set of `ActionConstructor`s that correspond to *Player* actions, and so describe all of the different different actions that a player can take, including a special `ForcedDiscardActionConstructor` for when players must make a choice of discarded card (which includes the use of an *Event Card* rather than its discard). `Action`s generated by these `ActionConstructor`s can be handed off to agents for consideration or use on a copy of the game state, but agents are never able to directly affect the game state held by the `GameLogic`. They must hand the `GameLogic` an `Action` to apply itself.

Item (c) points to the set of `ActionConstructor`s that correspond to non-*Player* `Action`s, which includes those required during the *Player Draw* and *Infect Draw* phases of the game. These `Action`s are each attached to a card (or cards in the case of an *Epidemic*) that might be drawn from a deck, and its effect on any `Board` corresponds to the effect of drawing that card. An agent can request these `Action`s for a `Board` in order to construct potential game histories, just like they can request *Player* `Action`s in order to find potential lines of play.

3.1.2 Game Logic Methods

Central to the design of the `GameLogic` are the methods that allow it to serve as an intermediary between the `Board` and an agent, as well as implement rules about the phases and logic of the game that aren't captured at the `Board` level. I want to describe those methods here. Some of the methods are illustrated in pseudocode algorithms below.

- `is_terminal(Board)` returns whether or not a given game state is one of won, lost, or broken. A `True` indicates that no further transition is required and the game is over.

- `is_stochastic(Board)` returns whether or not a given game state requires non-player transitions to be performed by the game logic. If a game is terminal, it will return *false*
- `get_stochastic_action(Board)` returns a potential stochastic `Action` (i.e. a potential card draw) from the given game state, using the non-player `ActionConstructor`s
- `nonplayer_actions(Board)` is a method that advances the `Board` in-place according to transitions that are outside of the players' control. This specifically includes card draws at the end of a player turn until either (a) a player is required to discard a card, (b) it is now the beginning of a player turn, or (c) the game has ended. The simple pseudocode is shown in algorithm 5.
- `random_action_bygroup(Board)` generates a random available player action for the current game state and implicitly encodes the logic of the default policy. It chooses a random *type* of legal action based on the legality of the `ActionConstructor`s, and then returns a random action of that type. One can see in algorithm 7 that it (a) it always returns a Cure `Action` when mild conditions are met and it is legal, and (b) only considers a `DoNothingAction` when neither treating or curing are legal.
- `list_actions(Board)` returns a `vector` of all legal actions available in the given game state.
- `rollout(Board,Heuristic)` takes a game state and uses `random_action_bygroup` and `nonplayer_actions` to traverse the game to a terminal state, then uses `Heuristic` to return a reward from the rollout. The psuedocode is shown in algorithm 6. On a 2.7Ghz Intel Core i5 (Mac OSX) CPU, this method is capable of running about 15,000 random full-game simulations per second.

Algorithm 5: Advance the game with non-player actions (in place)

```
Function nonplayer_actions(Board board):
    while is_stochastic(board) do
        action = get_stochastic_action(board)
        action.execute(board)
    end
```

Algorithm 6: Play a random game and return heuristic value of the final state

```
Function rollout(Board board,Heuristic h):
    while not is_terminal(board) do
        nonplayer_actions(board)
        if not is_terminal(board) then
            action = random_action_bygroup(board)
            action.execute(board)
    return h(board)
```

3.2 Agent Implementation

Agents are responsible for choosing an action and handing it to the `GameLogic` in order to advance the state of the game. All agents are responsible for having a `take_step(Board)` method which will be called to get them to make a move, and a `generate_action(Board)` method with into which they can incorporate all of their decision logic.

All agents are instantiated as a child of a pure virtual `BaseAgent` with these two virtual methods.

Most of the agents in the codebase are search agents that require special tools for building and evaluating a partial game tree. There is one uninformed random agent used to characterize the default policy, which takes a `random_action_bygroup` at each step.

Because the search agents require so much description, I'll break it into two parts. In one part, I'll describe the shared set of entities and algorithms that are used by many agents. Then I'll describe the agents themselves and which combinations of component parts make them up.

Algorithm 7: Generate Random Action (by type of action, then legal action of that type)

```
Function random_action_bygroup(Board board):
    if A player must discard a card then
        /* Always return a discard action if it is required       */
        return ForcedDiscardConstructor.random_action(board)
    if It is legal to cure then
        if No disease has ≥ 18 disease cubes then
            if There's 3 or less outbreaks then
                /* Always return a cure action if it is legal and some
                   conditions are met                                */
                return CureActionConstructor.random_action(board)
    /* Set the number of available player actions:  count DoNothing
       only if both Treat and Cure aren't legal                      */
    n = Number of Player Action Constructors
    if Both Cure and Treat aren't legal then
        n++
    i = Random integer   mod  n
    while True do
        if i==n then
            /* If DoNothing is legal and we chose it, return such an
               action                                                */
            return DoNothingConstructor.random_action(board)
        else
            if Player Constructor i is legal then
                /* If the ith constructor is legal, have it generate a
                   random action                                     */
                return PlayerConstructors[i].random_action(board)
            else
                /* Otherwise choose a new random number             */
                i = Random integer   mod  n
```

3.2.1 Search Tree

The core of this project is comprised of the search agents, and at the core of each search agent are the tools that allow them to build and evaluate search trees. In this section I'll describe these tools and objects.

The most important and basic element of the implementation are the `Node`s of the

search tree. There are two kinds of nodes present in the tree: `StochasticNode`s and `DeterministicNode`s. A `DeterministicNode` corresponds to a node of the game tree which requires an agent choice of action, while a `StochasticNode` corresponds to a chance node of the theoretical game tree. Both of them are children classes of a pure virtual `Node` class, and so can be cast as such.

Every `Node` has an `int` number of visits and `double` total observed reward, as well as boolean flags for whether or not it corresponds to a terminal and/or stochastic game state. Further, every `Node` has a `Node` parent and a container of `Node` children. Most importantly, every `Node` maintains the `Action` that was used to get to it from its parent, and a numerical `score` corresponding to its current node score (e.g. UCB1). `DeterministicNode`s also keep an `action_queue` defined on its instantiation that stores available `Action`s from the corresponding game state. This queue is used to create new children during tree construction.

The remainder of `Node` details lie in their methods, which include functions for the retrieval of attributes, children, or parent, as well as updating their node `score`. Perhaps the most important method for this project is `best_child`, which returns a child depending on whether or not the `Node` in question is stochastic or deterministic. In both cases under this implementation, the function transitions a board in-place using the action held on the child. The `best_child` functionality for deterministic `Node`s is described in algorithm 8. For stochastic `Node`s, the child selection is random, and its children have to be instantiated by a search tree rather than itself[7].

With basic `Node` in mind, we can introduce the notion of a tree. In this implementation, I call the tree entity `KDeterminizedGameTree`, whose name indicates that (a) it is parametrized by the number of determinizations one wants to make, and (b) It represents a partial game tree. It is instantiated with both a set number of deter-

[7]Unlike deterministic nodes, the exact behavior of stochastic transitions has to be determined from above. Stochastic nodes otherwise have no inherent idea of how many siblings it has or how many children it should make

Algorithm 8: Deterministic Node best_child

```
Function best_child(Board board,GameLogic game_logic):
    if not action_queue.EMPTY() then
        action = action_queue.POP()
        action.execute(board)
        if game_logic.is_stochastic(board) then
            return New stochastic node corresponding to board and action
        else
            return New deterministic node corresponding to board and action
    else
        best_child = arg max_c{c.score : c ∈children}
        action = best_child.get_action()
        action.execute(board)
        return best_child
```

minizations, and a `GameLogic` attribute that it uses to evaluate game states during tree construction[8]. An illustration of the tree is shown in figure 7. In order to traverse the tree, there are three basic steps. First, a random determinization k is chosen ((a) in the figure). Secondly, a copy of the current board state ((b), tracked on the right) is created. Then, starting at the root, deterministic `Node`s are traversed with `best_child` until a stochastic `Node` is reached ((d) in the figure). At this point, the kth stochastic child is selected. Thereafter the stochastic children are traversed sequentially (each corresponding to a card draw) until a deterministic or terminal `Node` is reached. After that point selection resumes as before until a new deterministic node is created.

Importantly, the copy of the board is altered in-place with each transition down the constructed search tree, resulting in a direct correspondence between the final state and the newly generated tree `Node` ((e) in the figure). This altered copy of the original game state can then be rolled out or evaluated.

This implementation ends up recreating the exact notion of a determinized game tree, with the exception that in implementation stochastic nodes can have stochastic children. Importantly, however, there are only ever exactly K histories, each of which

[8] In the following descriptions of tree-search methods, this attribute is what is called

corresponds to a single transition between deterministic nodes.

One important technical note on the notion of determinizations is that they are necessarily *shared*: the kth history is the same on every part of the tree. In order to enforce this behavior and also save on space and time, my `KDeterminizedGameTree` keeps a `determinization_queue` that stores K sequences of potential card draws that are referenced across the tree. Every time a stochasticity has to be resolved in the construction of the tree, the tree checks whether or not there are enough actions in the `determinization_queue` to traverse it to the next `Node`: if there are, then it will use those, otherwise it will collect new actions from the `GameLogic` and store those in the queue.

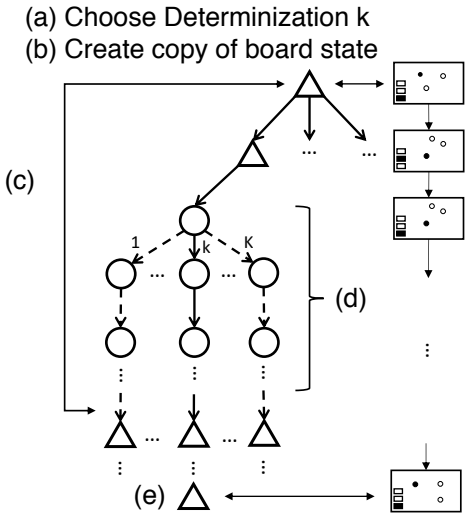


Figure 7: A KDeterminizedGameTree reflects the theoretical structure of a determinized game tree, with some notable changes (discussed in main text)

The overall behavior of game tree traversal is shown in algorithm 9. One important note is the the behavior of `getBestLeaf` will always return a deterministic node, even if a stochastic node was just created. The intention is to provide MCTS with an exact correspondence to the theoretical stochastic game tree (constrained to K determinizations), wherein every deterministic choice results in a deterministic node. Further, in this implementation it is possible that a stochastic node has only defined a fraction of the K possible child sequences, since construction is done lazily upon traversal.

Algorithm 9: The tree traversal getBestLeaf() KDeterminizedGameTree method and its functional dependencies. Encapsulates the Tree Policy.

```
Function getBestLeaf(Board board, Game Logic game_logic):
    best_choice = ROOT.best_child(board,game_logic)
    D = random determinization
    while (not best_choice.terminal) and best_choice.N_visits> 0 do
        if not best_choice.stochastic then
            best_choice = best_choice.best_child(board,game_logic)
        else
            best_choice = GetDeterministicChild(best_choice, board, D)
    return getDeterministicChild(best_choice, board, D, False)
Function getDeterministicChild(Node n, Board board, int D, bool
 on_sequence):
    if (not n.stochastic) or n.terminal then
        return n
    else
        if n doesn't have its immediate children then
            if on_sequence then
                Give n one null child
            else
                Give n D null children
    if on_sequence then
        i=0
    else
        i=D
    if ith child exists then
        n = ith child
        action = n.ACTION
        action.execute(board)
        return getDeterministicChild(n, board , D, True)
    else
        n = ith child                                  /* Will be null child */
        n.ACTION = GetOrCreateAction(n, board, D)
        n.ACTION.execute(board)
        return getDeterministicChild(n, board, D, True)
Function GetOrCreateAction(Node n, Board board, int D):
    if determinization_queue[D] has enough actions then
        return Next appropriate action
    else
        action = game_logic.get_stochastic_action(board)
        Add action to determinization_queue[D]
        return action
```

3.2.2 Search Agents

There are only a few kinds of search agents built in this project. Most of the variation between agents is due to the type of heuristic that they use in order to evaluate either the leaf game state or rolled-out game state.

Algorithm 10: Main loop for MCTS agents in generate_action

```
1  Function generate_action():
2      tree = new KDeterminizedTree(K)
3      sims_done = 0
4      while sims_done< N do
5          board_copy = game_logic.board_copy()
6          best_node = tree.getBestLeaf(board_copy)
7          reward = game_logic.rollout(board_copy, Heuristic)
8          best_node.backprop(reward)
9          sims_done++
10     Update Agent Measures
11     return Best root action
```

All agents are parametrized by both K, the number of determinizations to be made at each stochastic node, and N, the number of game simulations to be used for a decision (whether or not each is played fully to a terminal state). In addition, all search agents have a `GameLogic` attribute which they can use to make queries for actions and board information.

Algorithm 10 shows the primary decision loop for all search agents. The loop starts with line 5 which corresponds to item (b) of figure 7. Then the tree is traversed, advancing the board copy in-place, according to `getBestLeaf` shown in algorithm 9, within which a determinization is chosen. Not shown in this algorithm is an optional argument for a choice of selection criteria (e.g. UCB1 with a different exploration constant), which is not varied in this project.

In line 7, rollout agents hand a heuristic to the game logic with which it will evaluate a rolled out game state and return the resulting number. This line represents the

primary bifurcation of agent types. In heuristic-guided UCT agents, this line is replaced with a direct application of a choice heuristic (`reward = Heuristic(board_copy)`). One type of agent uses a heavy (i.e. informed) rollout that evaluates potential successor states and makes an epsilon-greedy choice.

Lastly, in line 11 comes the *selection policy*: how the agent takes an action based on the constructed game tree. Many of the MCTS agents use the canonical selection policy: taking the action with the highest UCB1 score. All other agents calculate the Expectimax value for each child and take the largest ("Greedy Expectimax"). The only modification to this expectimax calculation is that the expected node reward (`TotalReward/N_visits`) is returned by deterministic nodes that have not expanded all their children. In addition, stochastic nodes always return an equally weighted average of children deterministic nodes, since each is equally likely to occur, no matter how many times each was traversed.

3.2.3 Heuristic Definitions

Since the primary determinant of an agent is the heuristic it uses to evaluate states (whether or not they're rolled out), it is important to define them clearly. All heuristics are a function that returns a number in $[0, 1]$ given a `Board`. Here I provide a list of the heuristic functions I use and, where important, exacting definitions of their calculation.

- The simplest heuristic is just a wrapper for the win/loss status of the game. It provides no additional information. It is 1 when the game is won, and 0 when lost. It cannot be used on non-terminal states. I refer to this heuristic as "None" or "Naïve" since it has no additional information.

- The next simplest heuristic measures the fraction of diseases cured, and so takes the values $\{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$, where the value 1 is returned only for winning states. Because disease cures represent disjoint subgoals of the game objective, I refer to this as the "Fraction Subgoals Satisfied" heuristic.

- Continuing with the notion of tracking the proximity to satisfying winning conditions, I made an informed heuristic that measures the fraction of diseases cured plus the fraction of preconditions towards the *Cure* action that are met in the current state for each uncured disease. This formulation is directly inspired by classical planning heuristics. This heuristic is always at least as much as the "Fraction Subgoals Satisfied" heuristic, though does not come with the guarantee that it is monotonic over the course of the game. The calculation is shown in algorithm 11 and is designed to make the Cure action at least as important numerically as satisfying any preconditions, and that agents are rewarded for satisfying more Cure action preconditions. N_d is the number of cards held by player p of the same color as disease d, and R is the number of cards required to cure by player p.

Algorithm 11: Cure Preconditions Heuristic.

```
Function CurePreconditionsHeuristic(Board board):
    value=0
    for each cured disease d do
        value = value + .25
    for each uncured disease d do
        frac_cards = max_p [min(1, N_d / R)]
        closest_player = arg max_p min(1, N_d / R)
        value = value + .15 * frac_cards
        if frac_cards==1 then
            if closest_player is at a research station then
                value = value + .05
            else if closest_player is next to a research station then
                value = value + .025
    return value
```

Heuristics can also measure the proximity to losing rather than winning games. This is most important when evaluating the leaf states of the search tree rather than rolled out (and therefore terminal) states. I created two heuristics designed to reward agents for tackling elements of the game that will induce a loss, mainly because of preliminary

experimental results that suggested this might be necessary.

- The most direct heuristic for measuring "closeness" to losing directly evaluates attributes of the game state. It calculates both $\frac{\text{outbreaks}}{8}$ ("`outbreak_badness`") and $\max_d(\sum_{\text{City } c}\text{disease\_count}(d, c))$ ("`max_disease_badness`") then returns

$$1 - \max(\texttt{max\_disease\_badness}, \texttt{outbreak\_badness})$$

  (and also clipped to [0,1]). This calculation is a direct measure of board attributes that cause an agent to lose: the closer they get to a loss-inducing value, the the lower the heuristic value gets. I call this "Loss Proximity" or "Uninformed Loss Proximity".

- I also created a loss-proximity heuristic encoded with my own knowledge of the causal nature behind losing attribute statuses - specifically that cities with 3 disease cubes are *much* worse for the game outlook than cities with less disease. The creation of this kind of heuristic valuation seemed crucial after initial experimentation showed that agents could not successfully identify which "bad" board elements were worth addressing.

  The logic is described in algorithm 12. Note that it takes into account whether or not a city will *ever* be drawn again, and so allows for very late-game decisions to take advantage of the fact that some highly-infected cities are no longer at risk of outbreak. In addition, a consideration of neighboring cities allows for potential chain outbreaks to be considered worse than two distant cities both having three disease cubes. While a single city with three disease cubes gets valued at 1, two next to eachother get valued at 3, and three next to eachother would get valued at 6. This factor of 6 helps to inform the final normalization of value sum by 12. This comes from the assumption that under normal rules of play, the total value_d of any one disease is unlikely to be at least 6, and in fact if one is more than it

almost certainly means that at least one other is close to 0. So the normalization of $\frac{1}{12}$ essentially assumes that no more than two diseases can be in a disastrous state.

I call this "Informed Loss Proximity" since it encodes implicit knowledge about the causal elements of the game.

The last kind of heuristic is a *Compound Heuristic* that can take any two heuristics and return a weighted sum of the two based on a given weight $\alpha \in (0, 1)$ according to:

$$\text{CompoundHeuristic}(\text{board}, H_1, H_2, \alpha) := \alpha H_1(\text{board}) + (1 - \alpha) H_2(\text{board})$$

This easy adaptation allows one to build agents that easily incorporate information from multiple heuristics.

Algorithm 12: Informed loss proximity heuristic.

```
Function SmartLossProximity(Board board):
    foreach disease d do
        value_d=0
    foreach City c do
        foreach disease d do
            if c has 3 disease cubes of color d then
                if c can ever be drawn from the Infect Deck again then
                    value_d = value_d + 1
                    foreach Neighbor with 3 disease cubes of color d do
                        value_d = value_d + .5
                else
                    value_d = value_d + .5
            else
                value_d = value_d + .25 * disease_count(d, c)
    value = 1/12 * sum_d value_d
    return max(0,1-value)
```

3.3 Experimental Design

At the apex of the codebase dependency chain are the `Experiment`s, which put together all of the components described so far in order to test the ability of agents in a game. Here I'll describe the implementation of `Experiment`s and the measurements that they collect, as well as a list of the experiments performed for this project.

3.3.1 Experimental Implementation

`Experiment`s are a class that completely encapsulates the notion of an experiment. Its attributes include:

- A fully specified `Agent` to be tested, which may or may not keep its own measurements during play.
- A `Scenario` that provides either a specific initial game state or a generative process for sampling from a distribution of initial game states. For example, a `Scenario` could always put agents into a position where it is possible they win within X turns, or a position in which they're required to employ some specific strategy to win.
- A **Number of Games** to be played by the agent in the given `Scenario`.
- A list of **Measurement**s to be recorded during the course of the experiment, for example the number of decisions made by the agent or the number of times a specific action is used.
- A file descriptor used to name the written file.

Because this notion of an experiment requires a full specification of all components of the experiment, it captures all of the dependencies of a single test of a single agent. This is beneficial because it provides for easy and completely modular replication. The downside is that repeated experiments and hyperparameter searches become much more onerous, and indicates room for expansion of the experimental implementation.

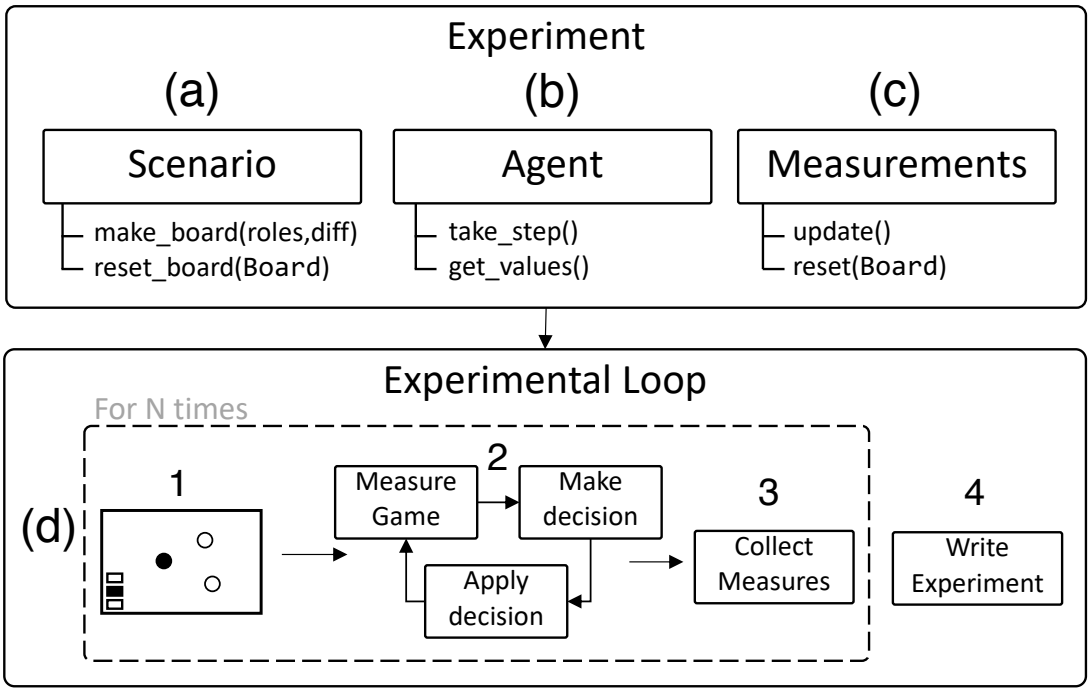


Figure 8: Illustration of the experimental attributes and the central experimental loop

Perhaps the most important attribute for this project are the `Measurements` . Every *Measurement* is a number that represents either some property of a played game, or some property of the agent that played it. This notion is implemented as a class (item (c) of figure 8) that maintains attributes necessary for tracking the game properties in order to derive the desired number, which are updated with an `update` method and collected with a `get_values()` method. They're segregated by whether or not they're collected by the `Experiment` (and so could be collected for any agent), or collected by the `Agent` (and so require internal information about the agent's decision making process). More precise definitions of the measurements collected during experimentation are given in the appendix.

In figure 8 one can also see a simplistic description of the experimental loop in item (d). The experimental loop is implemented as a function `RunExperiment(Experiment)` that plays games, collects measurements, and writes the corresponding results to a

descriptive header (`.header`) file and a game-level measurement `.csv` file. This function is called in a `main()` method defined within each `Experiment` C++ file.

At the beginning of the experimental loop, the `RunExperiment` function will first create or reset a current board (item (d.1) in figure 8 using the `Scenario` (item (a)) methods `make_board` or `reset_board`. At this point it also resets all of the measurement memory being kept by itself and the agent.

Next, in item (d.2) of the figure, the agent enters the game loop within the function, wherein the `RunExperiment` function goes through the iterative steps of `update()`-ing measurements and advancing the game state via agent `take_step()` or `nonplayer_actions()`. `Experiment`-level `Measurement`s are updated immediately before every `Agent` action.

Eventually, the game ends and in (d.3) the `RunExperiment` function will collect measurements from its existing measurements and, if applicable, the `Agent` that's playing. It will store these values in a growing string kept in memory. When all the games are played, this string is dumped into a `.csv` file and the `.header` file is updated with a final timestamp.

A pseudocode algorithm for the `RunExperiment` function is show in algorithm 13. In lines 2-9 the algorithm sets up variables for the main experimental loop. In lines 11-22, the function creates or resets both the `Board` and game measurements. Then in lines 23 through 28 a game is played to completion. In line 29 measurements are collected from the agent and the experimental game measures and put into the growing output string. When all of the prescribed games are played, the output string is written and the header is appended with a timestamp.

3.3.2 Experimental Process

All of the experiments in this project are the result of a number of games played on a board set up according to the regular rules of the game[9] and with the Medic, Scientist,

[9]With modifications described in the Background

Algorithm 13: Experimental loop.

```
1  Function RunExperiment(Experiment exp):
2      exp.write_header()
3      output_str = column headers from measurements
4      the_game = GameLogic()
5      the_agent = exp.get_agent(the_game)
6      game_measures = {}
7      if the_agent is measurable then
8          Add agent measurement headers to output_str
9      games_played = 0
10     while games_played< exp.n_games do
11         if There is no game board then
12             game_board = exp.make_board()
13         else
14             exp.reset_board(game_board)
15         if the_game has no game board then
16             the_game.reset_board(game_board)
17         if game_measures.EMPTY() then
18             game_measures = exp.get_game_measures(game_board)
19         else
20             foreach measure ∈ game_measures do
21                 measure.reset(game_board)
22         the_agent.reset()
23         while the_game is not terminal do
24             the_game.nonplayer_actions()
25             if the_game is not terminal then
26                 foreach measure ∈ game_measures do
27                     measure.update()
28                 the_agent.take_step()
29         Add measurements to output_str using get_values()
30         games_played = games_played + 1
31     exp.write_experiment()
32     exp.append_header()
```

and Researcher roles present (and in that order). With the exception of an experiment performed for the default policy with which 100,000 games were played, all were played with 100 games. All experiments were played with 4 epidemic cards in the deck, which corresponds to "regular" difficulty in the original game.

Further, all experiments were performed with a full host of measurements that measured aspects of the game (whether it was won, how many actions the agent took, how many times each action was used, how many outbreaks occurred, etc.) and agent decision making (how deep the average search tree went, what the score of the selected child was, etc.). These are used to derive all of the results shown in the next section. A full and more explicit list of measurements taken during experimentation, not all of which ended up being analyzed for this project, are included in the appendix.

Because the primary purpose of the project was to learn about, build, and test MCTS agents, the first experiments were done on canonical MCTS agents with only win/loss rewards from rollouts. Because a secondary goal of the project was to find the degree to which human-encoded state heuristics might improve performance, the next round of experimentation introduced new valuations on the terminal states reached in random games using heuristic functions. Due to the disappointing performance of all agents in these rounds of experimentation, more modification would be necessary.

In order to maintain the exploitation-exploration balance provided by UCT search, but recognizing that random rollouts may be the source of weakness in the first rounds of experimentation, the next rounds focused on replacing the rollout rewards with state value heuristics evaluated on leaf states of the tree. This would allow for a much more direct introduction of encoded knowledge and perhaps provide a stronger signal for directing policy. Experimentation with the same state rewards used for the last round showed that they were far too optimistic and ignored important elements of the game, which led to the testing of new kinds of state valuations.

Because heavy rollouts were not used in the original MCTS experiments, it was also worth it to see whether or not they provide a benefit when compared to the latest heuristic-guided agents. The intuition was that the same heuristics that judge leaf

states in the heuristic-guided UCT agents might make for local-search default policies that are much stronger than a random one.

The last rounds of agent experimentation were to perform some hyperparameter sensitivity tests and exploration, which could help show the degree to which performance in the best agents might be improved by modifying a few key parameters.

In addition to these agent experiments, I played 10 games with the same rule modifications as those experienced by the digital agents and recorded the win/loss status, the number of diseases cured, and decision depth (with the same definition as that used by agents) of each game.

The appendix holds a complete table of agents tested during experimentation.

4 Results

4.1 Game Characterization

Before exploring the behavior of any intelligent agents in the game, I first try to characterize the game by using a default-policy random agent. In addition to providing a description of the game tree, these findings establish a low-level baseline for performance. These results then provide guidelines for interpreting agent capabilities in several dimensions.

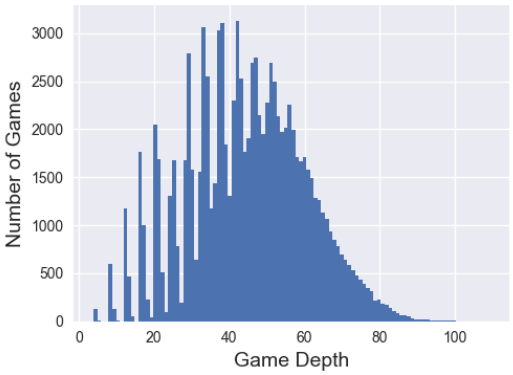
In addition I provide a human benchmark to serve as a weak upper bound on performance for these agents.

4.1.1 Default Policy Game Tree Traversal

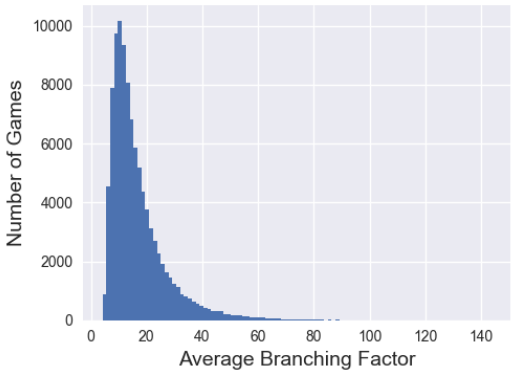
Initial game tree characterization using the default policy agent is show in figure 9. The game tree as explored by the chosen default policy has a mean decision depth of 44.6 ± 15.9. One can also observe the periodicity of decision depth frequency induced by the fact that player actions have to be made four at a time. This periodicity is made inexact since the use of *Event Cards* does not require the expenditure of an action, but does require an agent make a decision (and so increases depth). Players will occassionally be forced to discard a card too, which can happen during or outside of player turns and does not count as an action but does count as a decision.

The average mean branching factor across all 100,000 games is 16.8 ± 10.3, which one can see in 9b is approximately exponentially distributed. Lastly, figure 9c shows that the branching factor is inversely correlated to game depth, and directly correlated to the presence of high-branching *Event Cards* whose effect is amortized in longer games. This is direct evidence that these two *Event Cards* have an enormous impact on the branching factor observed by agents.

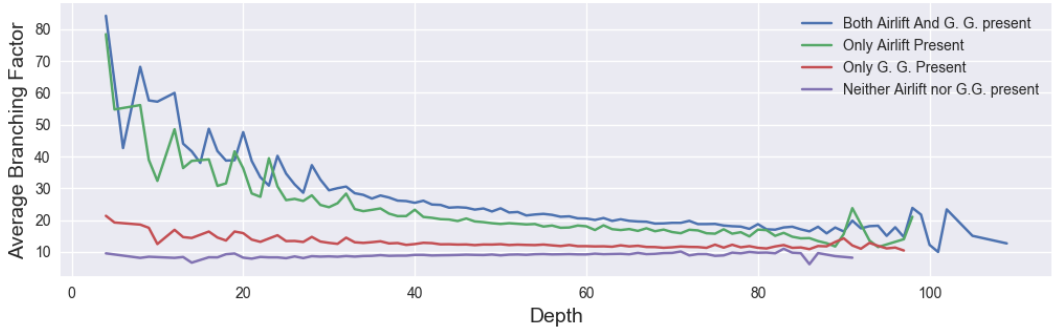
Without the presence of these cards, the branching factor is on average is a modest



(a) Game Depth Histogram



(b) Average Game Branching Factor Histogram



(c) Relationship between Branching Factor and Depth

Figure 9: Basic Game Tree characteristics observed from the perspective of a Random Agent in 100,000 games. Figure 9c shows that high branching factors are caused directly by the presence of the *Event Cards Airlift* and *Government Grant* (G. G.)

$9.0 \pm .7$ at each agent step. This is probably the safest estimate to use for branching factor, since (a) the cards might not be present at all ($\sim$ 24.0% of games), or (b) the use of *Government Grant* is restricted to the first player action ($\sim$ 25% of all nodes), (c) the use of *Airlift* is restricted to use at the beginning of a turn or after a player has reset their position memory (surely $<$ 50% of nodes). By rough estimate, then, this branching factor is an approximation for over 90% of nodes.

4.1.2 Default Policy Performance

Experimentation shows that the default policy wins the game in exactly 0% of 100,000 games. Further, results show that the default policy agent can only achieve one of the four sub-goals of the game (find the cure to a single disease) in $\sim$ 3.0% of games. It was able to find *two* out of four required cures in 36 out of 100,000 games, and in no games did it find more than two. These results suggest that it is fantastically unlikely that default policy is *ever* able to win the game, though only very unlikely that it is able to find any cure on its own.

Such findings underline the difficulty of successfully traversing the game tree to a win, and highlights the need to measure agent ability in more ways than how often it is able to win or lose.

4.1.3 Human Performance Benchmark

As part of experimentation, I also played 10 games in order to provide a human benchmark of performance according to the modified game rules. Nine of ten games were won, with an average game depth of 73.9, and 3.6 diseases cured on average (zero were cured in the one loss). Further, the one lost game ended at a depth of 8, which suggests that the upper limit of agent performance might be constrained only by the prevalence of unlikely and harmful random events.

4.2 Agent Evaluation

The incredible rarity of winning in random play makes it apparent that other measures of success need to be made for agents. Here I enumerate quantities that can be used to characterize an agent, ordered according to the notion that measures most closely tied to winning games matter more.

- **Winning Percentage** is obviously the most attractive measure to be able to maximize. More powerful agents should be able to win more often than others.

Evidence suggests that the most successful agents might not win more than 90% of the time, though surely far more than 0%.

- **Average number of diseases cured** is a direct way to measure how close agents got to winning. Because finding the cure for each of the four diseases comprise disjoint subgoals, this is a fairly good but not perfect measure of "closeness" to wins. An agent that myopically searches for a means to cure but otherwise ignores all indications of a looming loss might be able to cure more often than other agents, even though it isn't really performing well.

- **Game decision depth** is a measure of how long the agent plays the game without losing. Since a random agent plays to a depth of 44.6 ± 15.9, we'd look for agents to be able to play significantly deeper than this. Playing longer indicates that agents can successfully plan to mitigate the chance of losing even if there's no evidence that they can successfully move towards a win.

4.3 Agent Characterization

Here I characterize the decision making process for the two kinds of agents tested in this project: variations on canonical MCTS agents, and heuristic-guided UCT agents that replace rollout reward valuations with state heuristics. These results can be connected directly to later performance findings.

4.3.1 Search Depth

In this section I look at how deeply different agents search forward in the game. Hypothetically, this property reveals the extent to which agents are capable of incorporating deep plans into a choice of action. However, this does not necessarily indicate that decisions were made with a clearly measured reward difference or confidence, and so doesn't necessarily show increased power. Average search depth for each agent is described in

table 1. All average tree depths were between 8 and 11. In the context of the game,

Agent	**Heuristic**	**K**	**Simulations/Step**	
			10k	50k
MCTS	None	1	8.66	**10.1**
		3	8.09	9.69
	Fraction Subgoals Satisfied	1	8.68	**10.33**
		3	8.13	9.76
	Fraction Subgoals + Preconditions	1	8.90	**10.59**
		3	8.21	10.04
Heuristic UCT	Loss Proximity	1	8.89	**10.66**
		3	8.17	9.96
	Fraction Subgoals + Preconditions	1	9.03	**10.97**
		3	8.30	10.41
	Compound ($\frac{1}{2}$ Loss Proximity + $\frac{1}{2}$ Subgoals + Preconditions)	1	8.94	**10.67**
		3	8.22	10.11

Table 1: Table describing average mean tree depth across 100 games played by each agent. Bolded entries are the highest average depth for a given agent heuristic. For MCTS agents, "Heuristic" indicates the function with which terminal rolled-out states were evaluated. For Heuristic UCT agents, this was the function with which new non-terminal leaves were evaluated and used as a reward.

these depths would translate to planning at least as far as the end of the next players' turn (depth 8, when considered from the beginning of the active players turn), and at most into the next *next* players turn (depth 11, when considered from the beginning of the active players turn) on the most explored trajectory.

My main observation is that *more* simulations per step and *less* determinizations result in the greatest average tree depth. This is unsurprising, since on the one hand simulations per step controls the total tree size, and on the other less determinizations reduces the sampling requirement for repeated post-chance state visitations. One important aspect of this observation is that increasing the size of the game tree using simulations per step has a significantly larger impact than decreasing the number of determinizations from $K = 3$ to $K = 1$. I take this as an indication that trees are typically lopsided, since a tree that grows in depth almost uniformly would *not* increase its

average depth by over 1 without an exponential increase in the number of simulations per step. This lopsidedness may be an indication of some successful planning, but may also be due to the fact that even poor search agents will start constructing lopsided trees near the end of a game when lots of losing states are within the game tree horizon.

The fact that reducing the number of determinizations increases the average search depth aligns with the understanding that when $K > 1$, deterministic successors of a stochastic node can only be revisited and explored when (a) the same determinization is chosen at the beginning of tree traversal, and (b) the same deterministic predecessors are chosen for expansion according to the tree policy. This should cause any post-chance deterministic node to be visited K times less than its pre-chance deterministic ancestors. It is surprising, then, that the difference between $K = 1$ and $K = 3$ is as small as it is.

I believe this finding is mainly caused by the size of the full game tree compared to the size of the constructed one, together with the fact that trees are likely lopsided (see above). When most simulations are, on average, poured into the expansion of a relatively small segment of the tree, then the same set of chance nodes are being given roughly the same number of simulation passes, say N. Since the game is large past this chance node, N and $N/3$ have *almost* the same ability to traverse it deeply.

The amount of information in the heuristic also seems to have modest bearing on the search depth, though the effect is more pronounced with more simulations. I believe that this is because more informed heuristic evaluations (whether used to evaluate a rolled out or nonterminal state) help the agent to economize search depth for a fixed tree size much better, since it can make more discerning deep plans with less effort.

4.3.2 Selected Rewards

Table 2 shows the apparent reward of the action taken by the agent. As might be expected, a rollout-based agent receiving only win-loss rewards almost never sees the

Agent	Heuristic (Selection Policy)	K	Simulations/Step	
			10k	50k
MCTS	None (Highest UCB1)	1	0.00	0.00
		3	0.00	0.00
	None (Greedy Expectimax)	3	-	**.0004**
	Fraction Subgoals Satisfied (Highest UCB1)	1	.025	.036
		3	.029	**.044**
	Fraction Subgoals Satisfied (Greedy Expectimax)	3	-	**.21**
	Fraction Subgoals + Preconditions (Highest UCB1)	1	.30	**.33**
		3	.31	.32
	Fraction Subgoals + Preconditions (Greedy Expectimax)	3	-	**.52**
Heuristic UCT	Loss Proximity (Greedy Expectimax)	1	**.71**	.69
		3	**.71**	.70
	Fraction Subgoals + Preconditions (Greedy Expectimax)	1	.46	**.48**
		3	**.48**	**.48**
	Compound (50/50 Loss Proximity & Preconditions) (Greedy Expectimax)	1	.59	.59
		3	.59	**.60**

Table 2: Table describing average mean selected reward across 100 games played by each agent. Included are distinctions in selection policy upon exhaustion of the simulation budget. Highest UCB1 selected reward is the average backed up reward. Greedy Expectimax selected reward is the Expectimax reward value of the selected child.

prospect of winning, and so its selected reward is virtually always 0. There was one game in which a reward of 1 (game win) was observed in one action choice when that agent was using greedy Expectimax.

More generally, we observe greedy expectimax selection results in a much higher selected reward, on average, compared to UCB1 selection. This isn't surprising, since classical MCTS relies on making a choice based on all previous rollouts rather than the best-observed action sequence. Backing up expectimax values before selection forces a selection of optimal policy given the explored parts of the game tree, even if not representative of realized rewards.

Unlike in search depth, the number of determinizations and simulations doesn't seem to have a drastic impact on the difference in selected reward. The largest discrepancies

are observed in the agents using highest-UCB1 selection, which is perhaps a reflection of the fact that more powerful agents (those that can search deeper and derive lower variance choice-node value estimates) should, by construction, end up selecting children corresponding to higher valued rollout trajectories.

Further, the greedy-expectimax selection agents don't observe any meaningful differences for different number of simulations per step or determinizations. This might indicate that deriving more value from increasing search depth is difficult.

4.3.3 Selected Confidence

Agent	**Heuristic**	**K**	**N**	
			10k	50k
MCTS	None	1	-0.0%	-0.0%
		3	-0.0%	-0.0%
	Fraction Subgoals Satisfied	1	1.2%	2.7%
		3	0.8%	**2.8%**
	Fraction Subgoals + Preconditions	1	2.4%	**6.1%**
		3	1.8%	6.0%
Heuristic UCT	Loss Proximity	1	2.7%	**4.8%**
		3	2.1%	3.9%
	Fraction Subgoals + Preconditions	1	5.3%	**11.5%**
		3	5.0%	10.3%
	Compound (50/50 Loss Proximity & Preconditions)	1	3.6%	**7.5%**
		3	2.9%	6.4%

Table 3: Table describing the average (across 100 games played by each agent) mean percentage of visits that the selected action got in addition to a uniform simulation allotment over children. Bolded entries represent where this value is highest. A value of -0.0% indicates that the value was very small and negative.

One of the first things that this table can tell us is that an uninformed evaluation of rollouts for MCTS agents results in action selection that's effectively random on average. A small negative value indicates that, on average, the selected action has been visited *just* less than a uniform allotment of simulations over children. Together with information about selected reward in table 2, this tells us that this kind of agent is typically testing children one at a time in sequence repeatedly until it runs out of

simulations. When it runs out of budget, it returns the action that it would have chosen next for traversal.

One of the interesting results in this table is that increasing the number of simulations seems to *increase* the proportion of simulations that end up going into promising branches. This seems to indicate that somewhere between $10,000$ and $50,000$ simulations per step, the agent is better able to recognize that exploration terms on less-visited children aren't high enough to outweigh payouts in the most promising trajectories. After this tipping point, an increasing amount of simulations can be dumped in promising tree branches rather than using them to explore less-visited children.

An additional and sensible finding is that the proportion of visits going to the selected action beyond a uniform allotment goes *down* with an increasing number of determinizations. My intuition is that when $K = 1$, the agent can easily capitalize on the exact nonplayer-events revealed by the single determinization in order to make a consistent and deeper plan. When $K > 1$, courses of action have to be better *on average* over multiple determinizations, not just one, and so the agent is less often able to identify a clearly best course of action.

4.4 Agent Ability

In this section I describe the ability of each agent to play the game. As shown in the first part of this section, the game is incredibly difficult for a weak agent to play. As such, agent ability will be measured in terms of how often it wins games, how many cures to disease are found in each game, and how long the agent plays for. This section is separated into two parts: one describing the ability of MCTS agents, and one describing the ability of heuristic-guided UCT agents.

Selection Policy	Heuristic	K	N	Avg. Cures Found	Avg Depth ± std
Highest UCB1	None	1	10k	.04	42.5 ± 15.4
			50k	.01	46.4 ± 15.4
		3	10k	.00	45.6 ± 15.4
			50k	.03	45.1 ± 16.5
	Fraction Subgoals Satisfied	1	10k	.18	43.1 ± 14.6
			50k	.20	44.6 ± 16.0
		3	10k	.16	41.6 ± 13.7
			50k	.31	43.7 ± 15.7
	Fraction Subgoals + Preconditions	1	10k	.04	47.3 ± 14.0
			50k	.27	46.9 ± 13.5
		3	10k	.07	47.9 ± 13.1
			50k	.24	47.1 ± 14.8
Greedy Expectimax	None	3	50k	.16	50.7 ± 17.4
	Fraction Subgoals Satisfied	3	50k	.51	47.9 ± 17.5
	Fraction Subgoals + Preconditions	3	50k	.66	49.4 ± 15.8

Table 4: Table describing the ability of rollout agents according to their ability to cure disease and play the game longer over 100 games. None of the agents shown were able to win a game.

4.4.1 MCTS Agents

Table 4 shows summary statistics for the ability of MCTS agents. One of the primary findings here is that no canonical MCTS agent is able to win games. Further, a canonical MCTS agent that only backpropogates win-loss game rewards is indistinguishable from a random agent.

Importantly, some agents are able to cure disease at rates far above the rate of a completely random agent. This ability is increased both with the introduction of more detailed rollout-evaluation heuristics, and a more greedy selection policy. A more greedy selection policy has the largest effect on improving performance.

There is no significant difference in play depth between any agents. The mean play

depth seems to typically increase, though, with a larger simulation budgets and greedy selection policy. It does not seem to especially be affected by the information in the heuristic. In fact the "Fraction Subgoals Satisfied" agent also has the lowest mean depth values of all agents. This may have to do with the fact that the fraction of cures found is a non-decreasing quantity in any given game, and so rewards derived from this evaluation might provide a signal that allows the agent to occassionally make a depth-for-cures tradeoff.

Further, the "Fraction Subgoals Satisfied" agent does better under highest-UCB1 selection than the "Fraction Subgoals + Preconditions" agent in terms of curing disease in most cases. I expect that this is also a result of the fact that the fraction of cures found in any game history is a nondecreasing quantity, while the fraction of satisfied preconditions for uncured diseases may go up or down in subsequent random play. The fact that this high-variance effect seems to decrease performance highlights the weakness of random rollouts in providing value estimates to the agent. This negative effect disappears, though, when considering greedy expectimax selection on the same kinds of tree constructions, and so suggests that the high-variance effect of random rollouts with non-monotonic game state heuristics might be mostly nullified by re-evaluating the constructed game tree.

4.4.2 Heuristic-Guided UCT Agents

Table 5 describes the ability of pure heuristic-guided UCT agents. The most important finding is that some of these agents were able to win games. The uninformed and informed "Loss proximity" heuristic agents both have mean decision depths far above that of a random agent, though only the latter would be counted as statistically significant. In addition the uninformed loss proximity agent was able to cure disease at a rate comparable to an informed MCTS agent.

Because results are typically about as good or better with more determinizations

Heuristic	**K**	**N**	**Games Won (%)**	**Avg. Cures Found**	**Avg Depth ± std**
Loss Proximity	1	10k	0	.15	59.8 ± 21.3
		50k	0	.06	61.2 ± 18.7
	3	10k	0	.15	59.7 ± 17.6
		50k	0	.16	64.3 ± 20.9
Fraction Subgoals + Preconditions	1	10k	3	1.47	48.7 ± 14.3
		50k	4	1.40	46.2 ± 13.5
	3	10k	6	1.57	48.4 ± 16.4
		50k	6	1.39	48.2 ± 15.3
Informed Loss Proximity	3	50k	0	.02	100.7 ± 10.9

Table 5: Table describing the ability of heuristic-guided UCT agents (no compound heuristics) according to their ability to win the game, cure disease, and play the game longer over 100 games. The listed heuristic is that used to reward leaf states.

and simulations, experiments with compound agents were ran using $K = 3$ and $50,000$ simulations per step. The results of these experiments are shown in table 6.

The compound heuristic agents are able to win significantly more games, and win at most $\sim 30\%$ of games. Incorporating the informed loss-proximity heuristic provides the greatest boost to agent ability. I hypothesize that most of the utility of this addition comes in allowing for the player to prevent worse outcomes by taking early action, and so freeing the agent to myopically pursue curing for a much larger portion of the game. Weighing the informed loss heuristic less against the "Fraction Subgoals + Preconditions" heuristic helps to keep the agent less conservative in parts of the game where it has the freedom to do so, and end up curing and winning more often.

Table 6 shows the tradeoff between curing disease and lasting longer in the game. With increasing amounts of weight shifting away from informed loss proximity and towards "Fraction Subgoals + Preconditions", game depth decreases. Evidence shows that this tradeoff is worth making to some extent. Table 5 shows that the extrema of this weighting make for relative performance minima.

Because the most performant agents came at extrema of K and N, I also explored the

effect that increasing these values might have on performance. Table 7 shows the results of this search. Increasing the number of simulations per step has the most pronounced effect on overall performance, and in most cases is associated with a significant increase in the number of games won. The number of determinizations, on the other hand, has a less clear effect on agent ability. The percentage of games won goes up and down with increasing determinizations. $K = 20$ in particular seems a decidedly worse candidate for these values of N than any alternative. In all, these results indicate show that for any value of N there is likely to be a *best* value for K, and so yields a hyperparameter optimization problem. Further, these results suggest that measuring the fraction of games won may be a high-variance measurement.

Loss Heuristic	$\mathbf{w}_{\text{Precondition}}$	**Games Won (%)**	**Avg. Cures Found**	**Avg. Depth ± std**
Uninformed	1/2	6	1.58	56.3 ± 17.7
Informed	1/2	19	2.14	92.4 ± 16.7
Informed	2/3	29	2.52	80.1 ± 23.0
Informed	3/4	31	2.54	72.5 ± 21.3

Table 6: Table describing the ability of compound heuristic-guided UCT agents with $K = 3$ and $N = 50,000$ according to their ability to win the game, cure disease, and play the game longer over 100 games. All compound heuristic used "Fraction Subgoals + Preconditions" in combination with a loss heuristic and choice of weighting ($w_{\text{Precondition}}$).

K	**N**	**Games Won (%)**	**Avg. Cures Found**	**Avg. Depth ± std**
3	50k	29	2.52	80.1 ± 23.0
	100k	34	2.58	79.0 ± 20.9
5	50k	24	2.59	85.0 ± 17.6
	100k	39	2.79	79.9 ± 21.5
10	50k	34	2.52	81.27 ± 17.9
	100k	35	2.66	82.7 ± 21.6
20	50k	23	2.14	81.3 ± 22.8
	100k	28	2.59	85.0 ± 20.5

Table 7: Table describing the ability of compound heuristic agents ($\frac{2}{3}$ "Subgoals + Precondition" + $\frac{1}{3}$"Informed Loss Proximity") according to their ability to win the game, cure disease, and play the game longer over 100 games.

4.4.3 Effect of Heavy Rollouts

A comparison of a MCTS agent using a heavy rollout with a heuristic-guided UCT agent of the same power is shown in table 8. The heuristic-guided UCT agent used a compound heuristic that incorporated informed loss proximity for state valuation, and the rollout-based agent used the same compound heuristic for ϵ-greedy hill-climbing selection during rollout. Because the computational cost of this local search is much more than selecting a random action, the number of simulations per step was turned all the way down to $N = 500$ for both agents in order to make a fair comparison. Both agents used one determinization (K=1) in the hope that they could search as deeply as possible given the low simulation budget.

Agent	**Games Won (%)**	**Avg. Cures Found**	**Avg Depth ± std**
Heavy Rollout Agent	0 (%)	.25	46.9 ± 15.3
Heuristic-Guided UCT Agent	6 (%)	1.75	89.6 ± 21.3

Table 8: The heuristic-guided UCT agent uses the $\frac{2}{3}$ "Fraction Subgoals + Preconditions" + $\frac{1}{3}$ "Informed Loss Proximity" heuristic. An ϵ-greedy ($\epsilon = .1$) hill-climbing rollout is used for the heavy-rollout MCTS agent. The hill-climbing function is the heuristic used by the heuristic-guided UCT Agent. Terminal states are evaluated with a "Fraction Subgoals + Preconditions" heuristic for the MCTS agent.

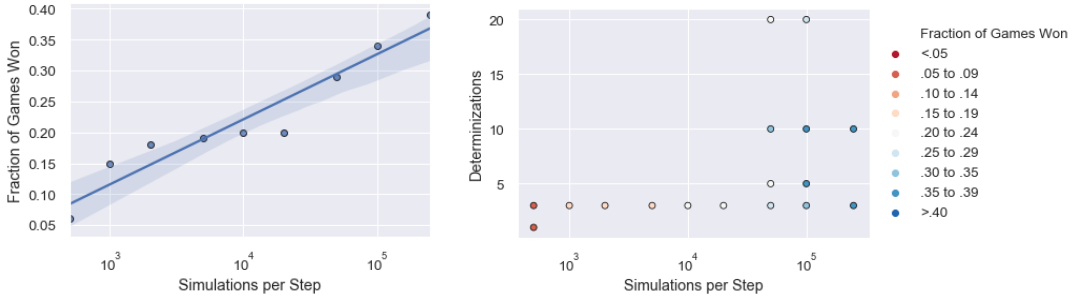
Heavy rollouts provide a clear advantage when compared to other the other MCTS agents. In fact, this heavy-rollout agent performs about as well as a $K = 3$ and $N = 50$k agent that uses a random rollout and highest-UCB1 action selection with the same terminal-state evaluation heuristic, even though it uses 1% of the simulation budget and less determinizations. This suggests that heavier rollouts help to provide the search tree a much better estimate of realizable state value on reward backups.

Despite the fact that heavy rollouts seem to improve MCTS markedly in that con-

text, a heuristic-guided UCT agent with the same power does much better. It does enormously better than any of the MCTS methods for any value of K and N, which highlights the central challenge of using rollout evaluations in the context of deep stochastic planning.

4.4.4 Effect of Simulation Budget

One of the most intuitive results shown above is that the simulation budget has a direct and consistent effect on the fraction of games won by the most capable agents. This effect can easily be demonstrated by figure 10a, in which one sees that an exponential increase in the size of the simulation budget results in a linear increase in the fraction of games won. Given the human benchmark, this suggests that achieving human-level performance might require as much as $\mathcal{O}(10^7)$ simulations per step. This would be prohibitively costly, and suggests that state-space or search reductions that maximize the efficacy of a fixed number of simulations might have the largest impact in this framework. Finding further heuristic optimizations might also be fruitful, though those effects are not thoroughly explored here.



(a) Effect of increasing simulation budget with $K = 3$.

(b) Performance of different values of N and K.

Figure 10: Exploring the effect of increasing the number of determinizations K and simulations per step N on win-rate for a $\frac{2}{3}$ Fraction Subgoals & Preconditions + $\frac{1}{3}$ Informed Loss Proximity heuristic-guided UCT agent.

Hypothetically, this finding could support the hypothesis that increasing N/K (simulations per determinization) in of itself is desirable. Figure 10b, which visualizes performance with a collection of (K, N) values for a specific heuristic-guided UCT agent suggests otherwise. It shows that while increasing N for any given K (and so increasing N/K) increases performance, it's not generally true that increasing N/K itself is desirable. At $N = 10^5$, for example, traversing from $K = 20$ down to $K = 3$ does not result in monotonically increasing performance as might be hypothesized. In fact the $K = 3$ agent performs worse than $K = 5$ and $K = 10$, even though N/K is larger. This further demonstrates that proper (K, N) selection is predicated on choosing an appropriate value of K for a desired budget N.

4.5 Agent Strategy

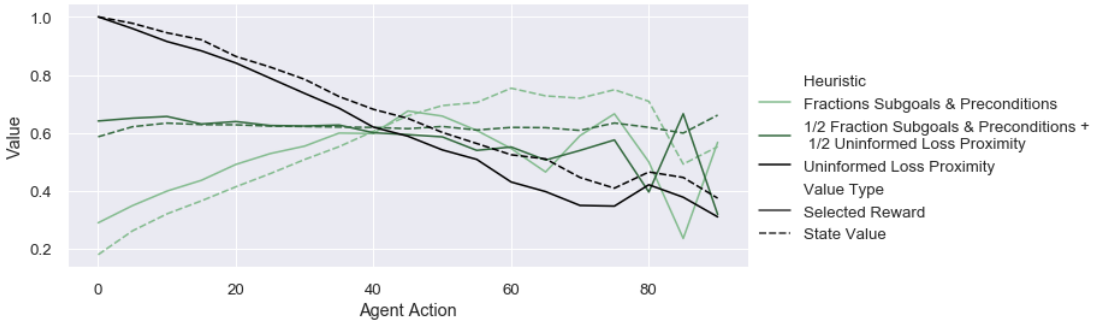
The goal of this section is to try to explicitly address part of one my research questions: What agent behavior corresponds to the best and worst performance? I approach this by using measurements taken during experimentation, and trying to evaluate what behavior makes for more successful play.

4.5.1 Optimism vs. Pessimism Tradeoffs

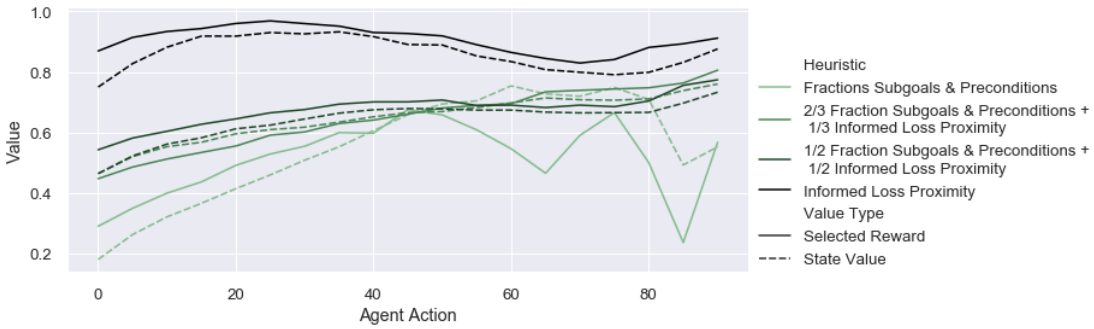
In this section I look at how the over- or under-estimation of successor state values relates to agent ability. Both experience with the game and qualitative observation of agent behavior has made clear that the ability to accurately weigh and act on risk is crucial to success.

Figure 11 shows trends in *selected* child reward versus current estimated *state value* as measured by the chosen heuristic, and for the former, determined by selection policy. For MCTS agents, the state value is determined by total backed up reward divided by the number of simulations. For heuristic-guided UCT agents, state value is the given heuristic evaluated on the current state.

(a) Average game trajectory of MCTS Agent selected versus state values



(b) Average game trajectory Heuristic UCT Agent selected versus state values (with naïve loss proximity measure and equally weighted compound heuristic)



(c) Average game trajectory of Heuristic UCT Agent selected versus state values (with crafted loss proximity measure and weighted compound heuristics)

Figure 11: Average game trajectory trends in state value versus selected reward, as measured by the chosen heuristic. All agents shown are $K = 3$ with N =50k simulations per step. Selection policy and heuristic both play an important role in determining the relationship between the two over the course of a game, and so the conservatism of agents.

In 11a one can see that the average selection and traversal behavior of MCTS (rollout-based) agents. The UCB1 selection policy (left) maintains a tight coupling between selected reward and state value. Unsurprisingly, the agent with no heuristic fails to see any value from any course of action, and traverses states that have no apparent value either. Otherwise, the close coupling of these values indicates that an agent can't discriminate between the value of successor states with any success, and so selected rewards are closely correlated to current state value. This inability squares with the performance observed for these agents. Further, one can see that by action 50-60 the agents take a decided turn for the worse in outlook and value, which represents the fact that all agents that have lasted this long are able to see inevitable defeat within the scope of their constructed game trees.

In figure 11a one can also see that using a greedy Expectimax selection policy (right) results in much more optimistic reward selections. This makes clear that *stronger* lines of play are being identified from the same trees that provide *weak* lines of play to highest UCB1-selecting agents. Being able to identify stronger lines of play has a clear positive effect on the growth of state value over the course of the game for both of the more informed heuristics, while an agent with no extra information still selects no higher reward and does not grow in value.

In 11b, one can see the value trends for heuristic-guided UCT agents. The loss-proximity heuristic guides the agent with pessimistic decision making: selected rewards are always lower than the current state value. This highlights the fact that the deeper into the game an agent goes (or searches), the "worse" the state of the game is inclined to get (since more *Infect Cards* and *Epidemic Cards* will be drawn). On the other hand, an agent guided by a "Fraction Subgoals + Preconditions" heuristic is initially slightly optimistic, but by the mid-to-late game the state value plateaus and the agent begins to

select successor states that are noticeably worse than the current one. This tells us that the agent is forced to pay later in the game for failure to prevent bad outcomes earlier. An agent guided by an equally-weighted sum of the other two maintains a closer value parity than either of the other two, though again by the end of the game is forced to select the least-bad successors.

Lastly, in 11c we observe a change when the more informed heuristic for loss proximity is introduced. Firstly, while the more informed loss-proximity agent is still slightly pessimistic, the added information clearly imparts an ability to stabilize the state value for much longer periods of the game. This is mirrored in the overall performance of the agent. When used in compound with an informed "Fraction Subgoals + Preconditions" heuristic, it introduces a stabilizing influence on the state-value trend that allows for the overall state value to grow more monotonically on average over the course of the game.

Together with results described in the section above, it seems apparent that having agents with a pessimistic outlook (who choose the best of an unattractive list of successor states) are crucial to making agents play longer into the game. At the other extreme, gains are only realized when an agent can capitalize on lines of play, perhaps under uncertainty, that move it closer to winning. This comes at the cost of ignoring how close it is to losing. The best agents incorporate both sets of information by using a compound heuristic to trace a strong line of play through the game.

4.5.2 Treating Disease

One of the central components of strategy and planning is treating disease on the board. Figure 12 shows the clear correspondence: agents that treat more last longer into the game.

In addition, one can see that agents guided only by proximity to loss (whether or

Figure 12: Game depth versus how many times the "Treat disease" action was used, with $K = 3$ and $N = 50$k for a selection of heuristic-guided agents, as well as the default policy agent. Lines represent the mean of game depth at each number of times that Treat was used.

not informed) treat more than agents that don't. The fact that the depth-to-treat trend for these agents is *below* that of the random agent suggests that they are being overly conservative, and treating more often than is truly necessary to achieve any given game depth. One can see, for example, that at depth 40 a random agent is treating less than 5 times on average, while an informed loss-proximity agent is treating about 10 times.

The trend for an agent that uses only a goal oriented focus ("Fraction Subgoals + Preconditions") is also shown. One can see that its trendline lies above that of a random agent, and that the agent treats disease significantly less often than the other agents. The fact that the trendline is above that of the other agents indicates that this agent is typically only treating disease when it is an immediate requirement for playing deeper into the game and being able to win.

4.5.3 Event cards and performance

Event Cards provide special and powerful utility to agents. I've found that (a) regardless of when the cards appear, the agent seems to use them within 3 actions virtually all of the time, and (b) *Event Cards* appear more frequently, on average, in winning games. The former finding indicates that agents are still probably not able to successfully utilize

Event Card	**Games Won in Presence (Games in Presence)**	**Games won in Absence (Games in Absence)**
Quiet Night	25 (79)	4 (21)
Government Grant	28 (85)	1 (15)
Airlift	25 (79)	4 (21)

Table 9: Table showing the relationship between the appearance of event cards and game outcome. Agent used for evaluation is $K = 3$ with $N = 50$k and a compound heuristic incorporating informed loss proximity with weight $\frac{1}{3}$.

the full power of these cards and instead use them for immediate tactical rather than long-term strategic gains. The latter finding primarily yields the unsurprising conclusion that drawing good cards makes it more likely to win the game. Table 9 shows how many games were won and lost in the presence or absence of each of the *Event Cards*.

Government Grant seemed to have, on average, the most positive impact on performance, as only 1 in 15 games without it were won, while about a third of all games in which it appeared were won. I would hypothesize that this is because while the effects of *Quiet Night* and *Airlift* appear once in the game, *Government Grant* introduces a permanent change to the board state by allowing the agent to place a new research station on the board. In addition, the building of a research station allows for both increased movement and an easier route to curing disease, which ties its use directly to winning the game. I further hypothesize that playing with an *Operations Expert* player role may result in winning more games, since the role can easily construct new stations on the map.

Quiet Night, like the other *Event Cards*, is used quickly upon its appearance. I believe that this is a *weakness* rather than strength of the agent. Because the effect is always positive (it always improves the game within the horizon of the game tree if used immediately), I suspect an agent is almost never able to identify a reason to save it. I would argue, though, that it becomes most effective when it can be used to either prevent otherwise unpreventable outbreaks in the immediate future, or when it can be used to free a player turn for trading cards or curing in lieu of addressing disease. In

both cases, one would likely have to save it for several turns before proper use.

4.5.4 Play Example

I also show here some example play exhibited by one of the more powerful agents, shown in figure 13. This game was played outside of an experimental setting, but was played on a board set up with the same scenario and with the same roles and difficulty.

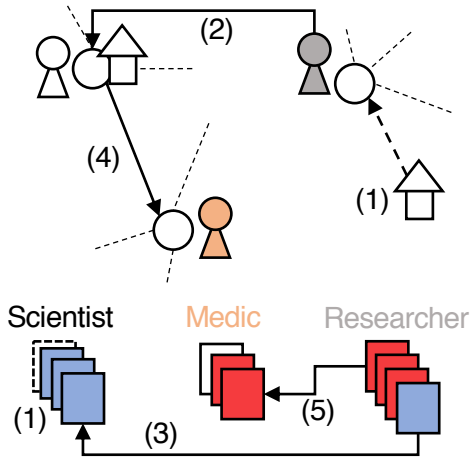


Figure 13: Diagram illustration tactical decisions on the *Researcher* turn. This heuristic-guided UCT agent playing used K=3 and N=100,000 and a compound heuristic comprised of $\frac{2}{3}$ "Fraction Subgoals + Preconditions" + $\frac{1}{3}$ "Informed loss proximity".

This illustration shows the sequence of actions, labelled sequentially, taken by the agent on the turn of the player with the *Researcher* role. This role gives that player the ability to freely give cards to other players as long as they're in the same city[10].

The sequence starts with the *Researcher* in a city far away from those occupied by the other players, and with no cards that would allow them easy movement to the region. The first action taken by the agent is to discard the *Government Grant* card held by the *Scientist* in order to build a research station in the city occupied by the *Researcher*. This does not expend an action. Secondly, the *Researcher* uses *Shuttle*

[10]Other roles require both players to be in the city that corresponds to the card being traded, which makes trading much more difficult

Flight in order to move to the city occupied by the *Scientist*, using the first action of the turn.

Third, the *Researcher* gives a blue city card to the *Scientist*, who has the ability to find a disease Cure with only *four* rather than the typical five cards of any disease color. This trade will allow the *Scientist* to immediately Cure blue on their next turn, since they're in a city with a research station.

Fourth, the *Researcher* moves to a neighboring city occupied by the *Medic*. Lastly with their fifth decision they give the *Medic* a red city card. This last trade leaves the *Medic* with three red cards, which combined with the two that they can take from the *Researcher* at the beginning of their next turn will allow them to Cure red during their next turn.

This action sequence highlights the ability of these agents to perform consistent tactical planning. The result of this plan is a guarantee that two Cures would would subsequently be found. Every decision, even though it was made under different determinizations, was able to identify and execute on the same optimal plan.

5 Conclusion

5.0.1 Summary

The performance of a canonical UCT agent is indistinguishable from that of a random agent in the deep stochastic planning problem represented by the board game *Pandemic*™. These agents are somewhat improved when win-loss terminal state rewards are replaced with state value heuristics that measure how close the terminal state is to winning, though performance remains very weak overall.

The final selection policy used after the construction of the search tree is shown to have a more dramatic effect on performance. As suggested in one paper (Björnsson and Finnsson 2009), the reward collected during search tree construction is *not* a good reflection of achievable reward, since the accumulation during construction might not reflect which actions are more likely to be taken in the future. Greedy Expectimax selection, specifically, is shown to dramatically improve the performance of MCTS agents by clearly identifying strong lines of play in the constructed game tree.

Heuristic-guided UCT agents show much stronger performance than their counterparts that rely on the results of random play. Even when converting random rollouts to ϵ-greedy rollouts based on a reasonable choice of heuristic, heuristic-guided UCT agents of the same power are far more performant. These findings present direct evidence that random game traversals provide a very weak estimate of state value in a complex planning problem, and so provide little useful information to an MCTS agent. Clearly, a direct estimate of state value provides a much stronger signal on which to base lines of play.

The best agents were able to capitalize on implicit encoded information about the causal relationship between specific board elements and their long-term effect on the game (i.e. the Informed Loss Proximity heuristic). This suggests that there is likely

a direct relationship between the searchable depth and performance, since agents that could observe and exploit those exact causal relationships would be able to make more informed decisions more consistently. This conclusion is further supported by the fact that increasing the size of the constructed game tree consistently results in better performance even with such informed heuristics.

It is also clear that the most successful agents require a direct accounting of not only the elements of the game that move them closer to winning, but move them away from losing, since no agent that only looked at one dimension of play performed very well. Heuristics that encoded more information about these game elements resulted in the strongest play.

Lastly, all agents performed far under the benchmark of human performance. This makes clear that other approaches to playing the game could make large improvements.

5.0.2 Ideas for future work

I believe that these findings make clear an offline learned valuation function, perhaps in combination with a policy-informed rollout[11] (a la Silver 2009), could perform better than any of the agents shown here. Such an agent could get around the simulation requirement for increased performance by exploiting knowledge from past games. In addition, the value derived from rollouts could become increasingly powerful as agent performance improves.

In lieu of offline learning, I believe a few modifications or explorations to the current framework could help improve agent performance. A primary area of interest is in collapsing the state space and perhaps even introducing some notion of graph search: after all, only the unique states at the end of each player turn have to be considered.

[11] Although evidence seems to suggest that these rollouts might not be necessary in this case

With further search reductions (perhaps even without), this could drastically improve search depth and performance.

In addition, one could alter the determinization method to always include at least the "best" and "worst" future events. The difficulty would be in providing the search tree a probability with which to weigh these events, let alone determining what is "worst" and "best".

Further, the heuristics used in this project went through very little tuning and sensitivity testing. It is entirely possible that small or modest adjustments to the weightings applied in the best-performing heuristics may have drastic impacts on performance. Future work could perform such a search in order to evaluate the sensitivity of performance with respect to the weighting of different board attributes.

One last modification that might be looked at is in the form of the Expectimax calculation. As-is, the calculation doesn't account for the fact that deeper states are simply more likely to have lower values, since drawing additional *Infect cards* can only reduce the value, while drawing *Player cards* might not increase it. To paint an extreme example one could suppose the agent were considering actions near the end of the player turn. The agent has explored action A and observed that it transitions through non-player actions that result in an unavoidable *outbreak*, which lowers the resulting Expectimax scores that are backed up. Sibling action B hasn't been explored, and its Expectimax value is not penalized even though the same outbreak *would* occur afterwards. The agent chooses action B simply because it hasn't explored it. It is possible that implementing discounts or tracking losing- and winning-status heuristics separately could help address this shortcoming.

There also may be room for the introduction of more classical ideas from search, game theory, or planning. The notion of *preference* as applied to changes in state could much more meaningfully distinguish similarly valued actions by evaluating what the

best action is based on its effect on each of an ordered set of preferenced attributes (E.g. Given that an agent can't cure any disease in the foreseeable future, actions should be taken to maximally reduce the chance of future outbreaks). In contrast to relying on a single number to capture all preference and forcing different priorities to be implicitly compared numerically, a preference might allow for more consistent and thoughtful planning to occur at different stages in the game. Multi-dimensional rewards may also be useful.

Identifying landmark *Subgoals* beyond curing disease, even if they are human-encoded, might also empower performance.

Lastly, there are many capabilities of my codebase which have been built but not used for this project. `Scenarios`, for example, would allow one to test and observe agents in specially designed situations. In this project I've only used a `Scenario` that reflects the setup of the original game. In addition, in this project I've only tested this "Vanilla" game scenario on a single difficulty and with a single set of player roles. I think other researchers or perhaps just enthusiasts might take interest in exploring the ability of different team compositions with different difficulty settings.

5.0.3 Pedagogical Reflections

This project provided a substantial challenge and learning opportunity. In a practical sense it has provided a very direct way to get experience designing and building my own system and tools from scratch. In this sense, that experience has provided me with a lot more confidence that large technical hurdles can be overcome with lots of work, careful thought, and attention to detail. In the same practical sense it has given me experience working on a timeline and keeping myself accountable for a certain level of quality and speed.

By building and testing these agents, I also feel that I have learned a lot about the

incredible difficulty of creating successful agents. As a student it seems easy to fall into the trap of believing that many real-world problems are easily solvable, since one is typically taught about the successes of the past and present rather than failures. It has also become obvious over the course of this project that the field of Artificial Intelligence is vast, and even having concluded my project I'm sure that there are better methods out there for the problem at hand. This leads me to the conclusion that true expertise in Artificial Intelligence comes in having a wide breadth of knowledge to draw on, and not in knowing the state-of-the-art methods of a single framework.

Lastly, the many failures I have experienced along the road to achieving mediocre performance have humbled me and helped me to recognize that there are a wealth of truly qualified experts out there who have gained their qualification with at least as much failure as success. I think that viewing success through this lense has helped to provide a much more human context for all of the content learned over the course of this year and especially this project.

Appendix

A Game Implementation

A.1 Map

The map is the only element of the codebase without any dependencies, and in some ways forms the bedrock of almost all of the logic for the game. I represent the map as a `namespace` that contains a few function definitions and `inline` variables that are referenced everywhere else in the codebase. It has a few key elements:

- An `inline` definition of each disease or city color as an integer:
  `BLUE=0, YELLOW=1, BLACK=2, RED=3`.
- An `inline` definition for the string names of each disease color, corresponding to their integer value.
- An `inline` definition of the 48 `CITIES` of the game, each of which has a hard-coded integer `index` in the range $[0, 47]$, a `string` name, `integer` color, `integer` population, and `vector` of neighbor indices.
- A few utility functions that can take an integer in $[0, 47]$ and return some attribute of the corresponding City.

These elements are one of the few places in the codebase where logic and attributes are hard-coded, because they represent the translation of physical game components into computer memory.

A.2 Game Board

The central component of my implementation is the `Board`, which is the logical representation of a game state. A figure that illustrates the representation of the `Board` is show in figure 14

In 14, one can see board attributes listed in item (a). Obviously, the board state tracks whether it is winning, losing, or has been broken by an inappropriate transition.

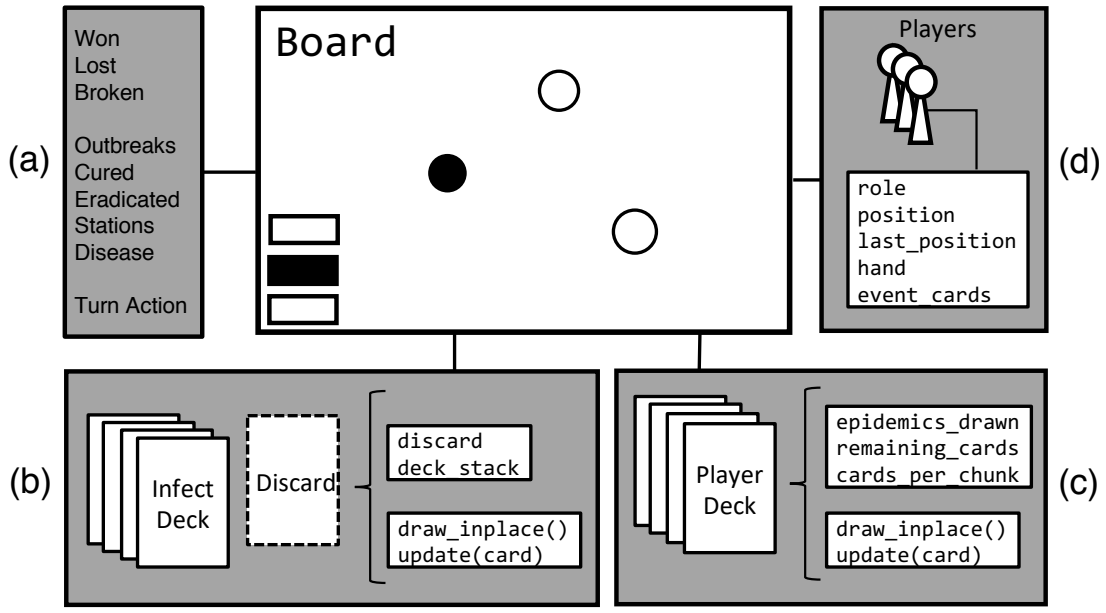


Figure 14: An illustration of board attributes. This board representation will be used in ensuing figures.

This last status was and is maintained actively throughout all of the experimentation and testing as a means to find logical errors. The `Board` also tracks game-level information like how many outbreaks have happened, which diseases are cured and eradicated, and where the research stations and diseases are around the map. Lastly, it tracks the phase of the game using a `turn_action`, which is incremented with each player use of an action, and with the end of each card-drawing phase, and so takes integer values in $[0, 5]$.

In the same figure one can find a brief illustration of the *Infect Deck* (item (b)), which I implement with an `InfectDeck` class. One can see that it tracks a `deck stack` as well as `discard`, which is important because the discard is intermittently place back on top of the deck.

The deck behavior is modeled with a push-down automata whose elements are collections of cards. The stack starts with a single element: the collection of all cards in the *Infect Deck*. A collection of discarded cards starts empty.

Whenever the deck is asked to generate a candidate draw via `draw_inplace`, it chooses a random card from the collection at the top of the stack. Every time the deck is asked to `update` its status for a candidate card, that card is deleted from the top collection of the stack and inserted into the discard collection. When the top collection is made empty, it is popped from the stack.

When the effect of an *Epidemic* is executed, a card from the bottom collection of the stack is drawn and added to the discard collection. Then, a copy of the discard collection is placed on top of the stack and the discard collection is emptied.

In item (c) of figure 14, one can see an illustration of the *Player Deck*, which is implemented with a `PlayerDeck` class. Like the *Infect Deck*, it is required to be able to generate potentially drawn cards with `draw_inplace`, and `update` its representation based on a drawn card. It keeps attributes that track how many *Epidemic* cards have been drawn, what non-*Epidemic* `remaining_cards` exist in the deck, and how many cards remain in each of the deck sections into which *Epidemics* were shuffled during setup.

Somewhat like the `InfectDeck`, it models deck behavior based on a push-down automaton. Each element in the stack is an integer representing the total number of cards left in the deck section on top of the *Player Deck*. As explained in the Background section, upon setup the *Player Deck* will be segregated into E equal sections, where E represents the number of epidemic cards to be played with. Each *Epidemic Card* is shuffled into a section and then the sections are stacked on top of eachother, with the largest sections on top in the case that it isn't equally divisible.

When asked to generate a potential draw, it checks to see whether an *Epidemic* has been drawn from the top deck section. If one already has, then it picks a candidate uniformly over all non-epidemic `remaining_cards`. If one hasn't, then with probability $\frac{1}{N}$, where N is integer value on top of the stack, it will produce an *Epidemic* card, and

with probability $\frac{N-1}{N}$ it will draw uniformly from non-epidemic `remaining_cards`.

When asked to update its status based on a previously generated candidate draw, it will first reduce the value at the top of the stack by one. When this value reaches zero, the stack is popped. If the given card was an *Epidemic*, it increments the `epidemics_drawn` by one. Otherwise, it removes the given card from the non-epidemic `remaining_cards`.

The last logical element of the `Board` are the players, simply illustrated in item (d) of figure 14. They are represented by a simple `Player` class whose attributes represent their current and last `position` as integers (corresponding to city `index`), their `role`, and both their non-*Event Card* `hand` and their `event_cards`.

In addition to these attributes, the `Player` class contains simple methods that will change a player position, add or remove cards from a players' hand, and return whether or not their hand is full.

There are a few key elements of the `Board` that are not illustrated in figure 14. Some crucial elements are the methods for performing game actions like infecting cities with disease, executing the logic of an *Outbreak*, and adding or removing research stations. Less crucial to the functioning of the game, but centrally important to the functioning of the codebase, are entrypoint methods that allow higher logic to request information from the board like who the active player is, where disease is on the board, and what cards might be drawn next.

B Experimental Details

B.1 List of Experiments

The table below is a list of all agent experiments performed for this project. All were tested over 100 games played with a Medic, Scientist, and Researcher role present (in

Agent Type	Heuristic	N	K
MCTS	None	10000	1
		50000	1
		10000	3
		50000	3
	Fraction Subgoals Satisfied	10000	1
		50000	1
		10000	3
		50000	3
	Fraction Subgoals + Preconditions	10000	1
		50000	1
		10000	3
		50000	3
MCTS (Greedy Expectimax Selection)	None	50000	3
	Fraction Subgoals Satisfied	50000	3
	Subgoals + Preconditions	50000	3
MCTS (ϵ-greedy Informed Compound rollout, Greedy Expectimax Selection)	Fractino Subgoals + Preconditions	500	1
Heuristic UCT	Fraction Subgoals + Precondition	10000	1
		50000	1
		10000	3
		50000	3
	Loss Proximity	10000	1
		50000	1
		10000	3
		50000	3
	.5 Loss Proximity + .5 Subgoal+Precondition	10000	1
		50000	1
		10000	3
		50000	3
	Smart Loss Proximity	50000	3
	.5 Subgoal+Preconditions + .5 Smart Loss Proximity	50000	3
	.75 Subgoal+Preconditions + .25 Smart Loss Proximity	50000	3
	$\frac{2}{3}$Fraction Subgoals + Preconditions + $\frac{1}{3}$Smart Loss Proximity	500	1
		500	3
		1000	3
		2000	3
		5000	3
		10000	3
		20000	3
		50000	3
		100000	3
		250000	3
		50000	5
		100000	5
		50000	10
		100000	10
		50000	20
		100000	20

that order). Each game was played with four *Epidemic* cards in the Player deck.

B.2 List of Measurement Definitions

As mentioned above, the goal of a `Measurement` is to derive a number that reflects some attribute of a played game. Here I enumerate and define the measurements collected during experimentation both by the `Experiments` themselves and by search agents. Not all of these measurements are analyzed let alone mentioned in the results section.

The measurements collected by the `Experiments` ran during this project are:

- `WinLose` is a measurement that returns 1 for games that are won, 0 for lost games,

and -100000 for broken games. It requires no updating and just reads the final board status.

- `LoseStatus` measures the attributes of a game state that are associated to losing status. It returns the number of outbreaks, the number of remaining player cards, and the amount of each disease present on the final board. It requires no updating and reads the final board status.

- `GameTreeSize` measures the decision depth and branching factor (number of decisions available per step) of traversed games. It returns the number of decisions made ("Depth") and the minimum, maximum, average, and standard deviation of stored branching factors. It updates a memory of observed branching factors at each agent step.

- `EventCardUse` measures the presence and usage of *Event Cards*. For each *Event Card* it returns the first decision at which the card was present (-1 if never observed), and the decision at which it was used (-1 if never used). It updates a memory of when and if each card has been used.

- `ActionCount` measures the number of times each *type* of action was used. It returns the number of times that each action was used over the course of a game. It updates a memory of counts associated to each type of action at each decision. Because it is updated before every agent action, it always fails to capture the *last* player action of the game.

- `CuredDisease` measures when and if each disease was cured. It returns the decision at which each of the four diseases was cured (-1 if never cured), and updates a memory of the status of each disease during each update.

- `EradicatedDisease` measures when and if each disease was eradicated. Like above, it returns the decision at which each of the four diseases was eradicated (-1 if never done), and updates a memory of the eradication status of each disease at

each update.

- `EpidemicsDrawn` measures how many *Epidemic* cards were drawn in the game. It returns only this number and requires no updating, and is only dependent on the final game state.
- `ResearchStations` measures how many research stations are on the board at the end of the game. It only returns this number and so requires no updating.
- `TimeTaken` measures the time taken between the instantiation (or resetting) of the measurements and the request for final game measurements. It only returns this number, and requires no updating.

Search agents used in this project maintain their own measurements, which are different because they have access to information made available during the decision making process of an agent. Because these measurements are not dedicated classes like those used in experiments, I don't describe them as logical entities. The measurements collected by search `Agent`s for this project are:

- *Tree Depth* measures the *maximum* search depth of constructed search trees during decision making over the course of a game. I measure the minimum, maximum, average, and standard deviation of *maximum* search depth over all decisions of a game. This requires the storage of maximum search depth at each decision step.
- *Selected Reward* measures the reward associated to selected child `Action`s over the course of a game. For MCTS agents with canonical highest-UCB1 selection, this is the average reward collected during tree construction. For any greedy-expectimax agent, it is the expectimax value of the chosen child. I return the minimum, maximum, average, and standard deviation of selected rewards collected over the course of each game. This requires the storage of the selected reward at each step of the game.

- *Selected Confidence* measures the upper confidence interval size associated to selected child `Action`s over the course of a game. I return the minimum, maximum, average, and standard deviation of selected confidence values over the course of each game. This requires storage of selected confidence intervals at each decision step.

- *Chosen minus Average Visits* measures the fraction of visits that go into the selected child `Action` compared to an equal allotment of simulations over all children. Values close to 0 indicate that simulations are being spread more or less equally across child actions, and little discriminates the chosen action from others. Higher values indicate that the agent is pouring a larger fraction of simulations into the exploration of the chosen action at the expense of sibling options. I return the average and maximum values for each game. This requires the storage of this value for each decision of the game.

- *Turn-level Selected Reward* measures the selected reward at specific decision intervals. I record the selected reward every 5 decisions, starting with the first and ending with the 90th. A value of -1 is returned for games that don't reach the required depth. This uses the memory of selected rewards used by the measurement of *Selected Reward*.

- *Turn-level State Value* measures the estimated value of the current state occupied by the agent at each step. In the case of MCTS agents, this is the accumulated average reward backed up over the course of search tree construction. For heuristic-guided UCT agents, this is a heuristic evaluation of the current game state using the same state-value heuristic as the search tree. I record the state value every 5 decisions, starting with the first and ending with the 90th. A value of -1 is returned for games that don't reach the required depth. This requires a memory of state valuations observed at each decision step.

References

Abramson, Bruce D. (1987). "The Expected-Outcome Model of Two-Player Games". AAI8827528. PhD thesis. USA.

Auer, P, N. Cesa-Bianchi, and P. Fischer (2002). "Finite-time Analysis of the Multiarmed Bandit Problem". In: *Machine Learning* 47.58 (2), pp. 235–256. DOI: https://doi.org/10.1023/A:1013689704352.

Bellman, Richard (1957). "A Markovian Decision Process". In: *Indiana Univ. Math. J.* 6 (4), pp. 679–684. ISSN: 0022-2518.

Björnsson, Yngvi and Hilmar Finnsson (Apr. 2009). "CadiaPlayer: A Simulation-Based General Game Player". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 1, pp. 4 –15. DOI: 10.1109/TCIAIG.2009.2018702.

Browne, Cameron et al. (Mar. 2012). "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4:1, pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

Chaslot, Guillaume M. J-b. et al. (Nov. 2008). "Progressive Strategies for Monte-Carlo Tree Search". English. In: *New Mathematics and Natural Computation* 4.3, pp. 343–357. ISSN: 1793-0057. DOI: 10.1142/S1793005708001094.

Coulom, Rémi (2007). "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". In: *Computers and Games*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 72–83. ISBN: 978-3-540-75538-8.

Gan, Xiaocong, Yun Bao, and Zhangang Han (Dec. 2011). "Real-Time Search Method in Nondeterministic Game — Ms. Pac-Man". In: *ICGA Journal* 34, pp. 209–222. DOI: 10.3233/ICG-2011-34404.

Ginsberg, Matthew L (2001). "GIB: Imperfect Information in a Computationally Challenging Game". In: *Journal of Artificial Intelligence Research* 14, pp. 303–358.

Howard, Ronald A. (1960). *Dynamic programming and Markov processes.* John Wiley, pp. viii 136 –viii 136.

Kearns, Michael, Yishay Mansour, and Andrew Ng (June 2001). "A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes". In: *Machine Learning* 49. DOI: 10.1023/A:1017932429737.

Kocsis, Levente and Csaba Szepesvári (Sept. 2006). "Bandit Based Monte-Carlo Planning". In: vol. 2006, pp. 282–293. DOI: 10.1007/11871842_29.

Lai, T.L and Herbert Robbins (1985). "Asymptotically efficient adaptive allocation rules". In: *Advances in Applied Mathematics* 6.1, pp. 4 –22. ISSN: 0196-8858. DOI: https://doi.org/10.1016/0196-8858(85)90002-8. URL: http://www.sciencedirect.com/science/article/pii/0196885885900028.

Leacock, Matt (Mar. 2020). "No Single Player Can Win This Board Game. It's Called Pandemic". In: *New York Times*. URL: https://www.nytimes.com/2020/03/25/opinion/pandemic-game-covid.html (visited on 06/18/2020).

Pepels, T., M. H. M. Winands, and M. Lanctot (2014). "Real-Time Monte Carlo Tree Search in Ms Pac-Man". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.3, pp. 245–257.

Santos, A., P. A. Santos, and F. S. Melo (2017). "Monte Carlo tree search experiments in hearthstone". In: *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 272–279.

Silver, David (2009). "Reinforcement Learning and Simulation-Based Search in Computer Go". PhD thesis. CAN. ISBN: 9780494540831.

Silver, David et al. (Oct. 2017). "Mastering the game of Go without human knowledge". In: *Nature* 550, pp. 354–. URL: http://dx.doi.org/10.1038/nature24270.

Swiechowski, Maciej, Tomasz Tajmajer, and Andrzej Janusz (2018). "Improving Hearthstone AI by Combining MCTS and Supervised Learning Algorithms". In: *CoRR* abs/1808.04794. arXiv: 1808.04794. URL: http://arxiv.org/abs/1808.04794.

Szita, István, Guillaume Chaslot, and Pieter Spronck (2010). "Monte-Carlo Tree Search in Settlers of Catan". In: *Advances in Computer Games*. Ed. by H. Jaap van den Herik and Pieter Spronck. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 21–32. ISBN: 978-3-642-12993-3.