ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/306058411

The States of Splendor: Searching Game Trees with Partial Information

Thesis · May 2016

DOI: 10.13140/RG.2.1.2268.4401

CITATIONS
0

READS
5,698

1 author:

Joshua Hepworth
Reed College
1 PUBLICATION **0** CITATIONS

SEE PROFILE

All content following this page was uploaded by Joshua Hepworth on 11 August 2016.

The user has requested enhancement of the downloaded file.

The States of Splendor: Searching Game Trees with Partial Information

---

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Joshua Hepworth

May 2016

Approved for the Division
(Mathematics - Computer Science)

Eitan Frachtenberg

Acknowledgments

First and foremost I want to thank Eitan Frachtenberg for advising me and helping to collect data on his improbably large computer. I hope you're able to continue teaching in the future.

Many thanks to Mom, Dad, Uncle David, and my late Grandpa Joe. I would not have found Reed, afforded it, or made it through without you.

Thank you Adam, Rachel, Elizabeth, and Dan for being my role models, blazing the trail, and offering your support and advice whenever I need it.

Thanks to the Reactor, Tir na Nog, and GameDev communities for shaping my Reed experience. And a special shout-out goes to Adrian and Max for introducing me to a host of board games including Splendor.

And finally, thank you Emma Rennie for playtesting my games, sharing my hobbies, and standing by my side for three years. You made me proud to be myself, and I can't imagine a better partner.

Table of Contents

List of Tables

List of Figures

Abstract

Space-searching algorithms are a critical component of board game AI. Such algorithms are well-studied in the context of historically popular strategy games like chess, checkers, and go. Less work has been done applying search algorithms to modern genres such as "German-style" games.

I designed six space-searching algorithms for the strategy board game *Splendor*. I used three of them as benchmarks, including a greedy algorithm, Minimax, and random search. Then I developed three genetic algorithms (GAs) and compared them against the benchmarks. Two of the GAs represented their strategy as a series of moves, while the third evolved a list of game goals.

The greedy algorithm performed surprisingly well against humans, although the number of games was too limited to draw inferences about win rate. Minimax and random search dominated the greedy algorithm, winning 80-90% of games with reasonable computational resources.

The move-based GAs proved difficult to exploit as they produced illegal strategies after recombination; they performed worse than random search. The goal-based GA outplayed Greedy and tied with RandomSearch, but lost to Minimax in 60% of games with equal turn time. The win rate improved when the search domain was expanded with additional goals, suggesting that further expansion would yield even better results.

The success of random search proves that deep searches can be effective in exploring Splendor's state space. However, the research suggests that moves are not a good building block for GAs. There is room for future work studying new GA representations and alternative search strategies which do not depend on representation at all.

Dedication

For Mom, who scouted every pitfall.

Introduction

0.1 Problem Description

Game-playing has been a focus of artificial intelligence since the early 1950's, when the first chess and checkers AI were invented. Turn-based games can be modeled as a tree where nodes represent game states, and edges represent legal moves. The size of a full game tree can vary immensely, from 10^{31} for checkers to 10^{123} for chess [1] to over 10^{700} for modern games [2]. The study of tree-searching techniques and the incentive to make competitive AI are therefore tightly tied.

Game AI has two components: offline and online. Offline processing refers to the development of a strategy that analyzes many different games and finding patterns. Online processing refers to evaluation that takes place during a game, such as searching a game tree. Breakthroughs in board game AI have tended to come from developments in offline processing, while tried-and-true tree searching algorithms like Minimax and Monte Carlo continue to be used for online processing. However, these algorithms are less effective under the complexity of larger search spaces.

Genetic algorithms have been shown to be an effective search tool in prohibitively large trees. [2] Our goal was to explore the use of genetic algorithms for online processing in modern board games. We designed our own implementation[1] of the popular board game Splendor and used it to compare three genetic algorithms, a greedy algorithm, and Minimax, a standard tree-searching algorithm.

Splendor is a turn-based board game centered on resource management. A German-style game[2], Splendor is potentially representative of a large class of games that have grown popular in the last two decades. Readers may be familiar with titles like Settlers of Catan, Carcassonne, Puerto Rico, or Ticket to Ride. While many of these games have an abundance of mechanics and unique features, Splendor is relatively simple.

Publisher Days of Wonder has released a digital version including three breeds of AI. Each follows a predetermined strategy, making it very predictable. This method is unfortunately common in smaller game companies which don't have the resources to develop and test expansive AI. Evolutionary algorithms could potentially be useful for making stronger AI on a reasonable budget.

[1] https://github.com/Zeplar/Splendor

[2] German-style games such as Splendor are defined by their emphasis on strategy and indirect player interaction rather than luck and conflict.

0.2 Contributions

We analyzed six tree-searching algorithms: Greedy, Minimax, Random Search, IndexMove, ExactMove, and BuyOrder. The latter three were genetic algorithms operating on different models of the problem space. Minimax performed well. The genetic algorithms outperformed Greedy, but failed to evolve well due to the lack of a stable crossover operator. Our result does not entirely rule out GAs, since there is a large space of plausible unexplored genetic representations. However, it suggests that Splendor and German-style games in general have no straightforward translation to GAs.

The data collected over the course of this project yielded some interesting tangential results regarding Splendor. Minimax players of different depths were pitted against each another. The strongest player beat the weakest over 80% of the time, demonstrating that strategy is important. Incremental depth gave diminishing returns, suggesting that luck in the card draw plays a significant factor.

Fitness functions were evolved for Minimax using a hill-climbing algorithm. The values assigned to these functions indicate the value of each board element, and may predict the importance of certain subgoals.

Finally, the performance of the starting player was recorded over many thousands of games. When both players used the same algorithm, the starting play won 60% of the time. This implies a moderate first-player advantage, an undocumented feature of the game.

Most board game designers do not make digital prototypes, which can be an expensive investment in a risky business. But that might be shortsighted. These statistics– the prevalence of luck, the predictive value of game pieces, and the starting player advantage– are all valuable information when balancing a game.

Chapter 1

Previous Work

The study of game-playing algorithms dates back to before modern computers existed. In the mid-to-late twentieth century computers were powerful enough to play chess, using algorithms like Minimax (discussed below) to search a few turns ahead. Early chess programs outperformed expectations, possibly because researchers underestimated the importance of breadth. One journalist wrote that chess programs won not by "playing well", but by noticing and capitalizing on human errors. [3] That is, a computer's thorough search was more likely to detect obvious blunders. Today, computers outperform humans in both breadth and depth of search.

1.1 Deterministic Games

Deep Blue

The most well-known game-playing algorithm is IBM's Deep Blue, famous for being the first chess program to defeat a grandmaster in the late 1990s. Deep Blue used a variant of Minimax along with a database of common openings and endgames. Moves were evaluated according to a complex heuristic with several thousand components. [4]

The development of Deep Blue focused on hardware optimization, tree pruning, and improvements to the scoring heuristic. The search algorithm itself was relatively simple. IBM's method required expert advice and thousands of game logs; while effective for chess, it would not be practical for modern games which do not have the wide following and rich history of older games.

AlphaGo

Another board game AI which made recent headlines is Google's AlphaGo. AlphaGo primarily uses Monte Carlo tree search, in which board states are evaluated by fully simulating many random games and returning the number of winning simulations. [5] Monte Carlo search has the advantage of not requiring heuristics at all, since games are simulated to completion. However, it is untenable for games like Splendor in

which progression of the game requires hidden information.[1]

Google's team also developed a neural network which selected starting moves for the Monte Carlo process. A preprocessing technique, neural networks require complete game logs to analyze; therefore they were not an option for Splendor.

1.2 Nondeterministic Games

Settlers of Catan

Szita, Chaslot and Spronck used Monte Carlo tree search to make an AI for Settlers of Catan. Their algorithm performed modestly better than other hand-coded programs, but could not compete with Szita in four-player games that included two hand-coded programs. [6] Settlers of Catan is widely regarded as the canonical German-style board game, so this result is promising for the use of search algorithms in generalized board games.

1.3 Genetic Algorithms in Nondeterministic Games

Genetic algorithms are a common search technique for problems with a complex fitness landscape. Previous work with genetic algorithms in the domain of nondeterministic multiplayer games has yielded some positive results. However, this work has taken the form of offline calculations used to produce an all-encompassing strategy. Offline calculations are limited because they cannot cover all possible games, and must generalize to some degree. Real-time calculations, on the other hand, are limited by computational resources. AlphaGo, for example, uses over two thousand CPUs and GPUs [7].

Wargus

Spronck and Ponsen [8] successfully developed an algorithm to evolve strategies for Wargus, a combat oriented real-time strategy game. The algorithm generated sequences of actions and selected from them as appropriate for the game state. The Wargus team encountered a problem inherent to stateful games: actions which are legal in one state might not be legal in every state. Their solution involved breaking the game into chunks depending on which actions were available. While the same problem was present in Splendor, the solution was not applicable.

Legal moves in Splendor are largely independent; for example, the ability to afford one card does not guarantee the ability to afford any other cards. Splendor is similar to other German-style games in this regard. Wargus, like many other real-time strategy games, has a strict dependency tree where game actions become available as the game progresses. This means that actions in Wargus can be grouped according to the stage of the game when they become legal, while moves in Splendor cannot.

[1] A possible method for employing Monte Carlo search in Splendor is discussed under Future Work.

The Wargus algorithm is promising for the applicability of genetic algorithms to complex games. However, the algorithm was only used offline to generate comprehensive strategies. Online evaluation consisted of implementing the devised strategy with the real-time analog to a tree searching algorithm.

War Games

Revello and McCartney [9] developed an algorithm to evolve strategies for a custom war game. The chosen game was a simulation of a naval blockade with complex rules and significant nondeterminism. Unlike most conventional games, the blockade simulator even had a probabilistic set of win conditions. Revello and McCartney set out to show that the genetic technique is a robust tool for generating competitive strategies for diverse and nondeterministic games. The paper found that genetic algorithms were effective in finding winning strategies, and that tuning the fitness function refined the solutions.

CIGARS

Louis and Miles used case-injected genetic algorithms (CIGARS) to plan strategies in a real-time strategy game. [10] These algorithms underwent normal evolution, but previous solutions to similar problems were occasionally injected into the population.

The case-injection technique cannot be applied to Splendor because chromosomes are highly specific to the board state when they were generated. Even disregarding the players' positions, there are trillions of possible board states.

Chapter 2

The Game of Splendor

Splendor is a turn-based board game centered on resource management. It was chosen for study because its rules were easy to implement, but define a sizable game tree with a branching ratio (20-25) comparable to that of chess. The description of the game is as follows:

Splendor is a game of chip-collecting and card development. Players are merchants of the Renaissance trying to buy gem mines, means of transportation, shops– all in order to acquire the most prestige points. If you're wealthy enough, you might even receive a visit from a noble at some point, which of course will further increase your prestige.

On your turn, you may (1) collect chips (gems), or (2) buy and build a card, or (3) reserve one card. If you collect chips, you take either three different kinds of chips or two chips of the same kind. If you buy a card, you pay its price in chips and add it to your playing area. To reserve a card (in order to make sure you get it, or, why not, your opponents don't get it) you place it in front of you face down for later building; this costs you a round, but you also get gold in the form of a joker chip, which you can use as any gem.

All of the cards you buy increase your wealth as they give you a permanent gem bonus for later buys; some of the cards also give you prestige points. In order to win the game, you must reach 15 prestige points before your opponents do.

([11])

Text is not the ideal medium for learning this game, but grasping the mechanics of Splendor is important for understanding the subsequent algorithms. In general, players spend the first few turns taking gems two or three at a time. These gems are used to purchase the cheap "Tier 1" cards, which give no point value but provide a single-color discount for subsequent cards[1]. The player's economy is thus nonlinear, but limited by available cards and the maximum of one action per turn. Nobles provide a late-game incentive for Tier 1 cards, since players may snag a noble while simultaneously improving their economy.

[1]A list of cards and nobles is supplied in Appendix A

Figure 2.1: Setup configuration for Splendor. From top to bottom: Nobles, tier 3, tier 2, tier 1, gems. Each card has a point value in the top left, a color identity in the top right, and a cost in the bottom left. Cards are replaced from the adjacent deck whenever they are bought or reserved. (amazon.com)

Chapter 3

Greedy Algorithms

3.1 What Are Greedy Algorithms?

A greedy algorithm always makes the choice that gives the best immediate result based on its evaluation heuristic. Greedy algorithms are very fast, because they don't have to compute multiple scenarios. However, a greedy strategy only works on problems with certain qualities. [12]

The first is the "Greedy-choice" property. This property means that a problem's optimal solutions do not depend on subproblems; therefore, the solutions can be found without solving the subproblems. The second property is "optimal substructure". This means that the optimal solution to the problem contains the optimal solution to subproblems. Optimal substructure makes it easy to prove that a greedy algorithm returns the correct answer. If the greedy choice is correct for the subproblems, then it is also correct for the original problem.

Splendor is nondeterministic and unsolved, so we're not worried about finding the true optimal strategy. In fact, it is likely that no strategy can win one hundred percent of the time. Instead, we use Greedy as a fast benchmark by which to compare other strategies. That is not to say that strategies are strictly ordered; some may have a particular advantage against others. But comparing every pair of algorithms and parameters would take months or years, and performing poorly against Greedy is an indication that a strategy is no good.

3.2 Splendor.Greedy

A greedy algorithm is not likely to be the best choice for a game like Splendor. The success of a greedy strategy would mean that the game is easy for humans, and its popularity would dwindle. Perusal of strategy forums like boardgamegeek confirm that this is not the case. However, such a strategy is easy to code and test, so it is worth exploring.

The Greedy player takes one parameter, a scoring function $F : Board \to \mathbb{R}$. The algorithm simulates and evaluates a board for every legal move, and chooses the move which generated the maximal board state When multiple moves are tied,

Greedy prioritizes in the following order: Buy, Take3, Take2, Reserve. Moves within a move type are always generated in the same order; this biases Greedy toward certain gem colors, but the algorithm performs well regardless.

A variety of scoring functions (heuristics) were designed, including the following:

- Prestige (Number of cards)
- Points
- Gems (Number of gems)
- WinLoss (1 for a win, -1 for a loss, 0 otherwise)
- Progress toward nobles this turn (1 card might apply to multiple nobles)
- Buys (Available legal buys)
- Lead (self points - opponent points)
- Turn (Turns into the future in a simulated game)
- allEval: $100 * WinLoss + 2 * Points + 2 * Nobles + Prestige + Gems$

The number of possible heuristics complicates the problem, since the performance of an algorithm depends on using the correct heuristic. In order to settle on a single scoring function, Frachtenberg ran a hill-climbing algorithm on the component heuristics. The result was allEval2:

- $(100 * WinLoss + 1.5 * Points + 2.5 * Nobles + Prestige + Gems) * 0.9^{Turns}$

In later algorithms we will see that simulating many turns into the future causes the simulated board state to become inaccurate, since new cards cannot be drawn in the simulation. Greedy is unaffected by this partial information, since it only evaluates a single turn. However, performance of other algorithms was vastly improved by the addition of exponential decay in turns-simulated.

Some of these heuristics are meaningless for Greedy. Lead always produces the same choice as Points, since the opponent's points are constant over the evaluation. Win/Loss is usually less informative than Points, since the player with the most points wins (ties are an edge case which comes up infrequently). Nonetheless, these functions are included here because they will be reused in every algorithm henceforth.

Scoring functions form an algebra– they can be added, scaled, multiplied, and even exponentiated. For example, the function (Points + 10*WinLoss) accounts for the edge case when buying a 0-point card during a tie breaks the tie for the opponent (ties go to the player with the fewest cards).

Greedy was originally given the Points function as the most obvious indicator of progress. Eventually, a hill-climbing algorithm was used to generate a stronger scoring function. Greedy(Points) prioritizes cards worth the most points or cards which will achieve nobles. Failing this, it buys any affordable card or takes the most gems possible. This algorithm is similar to an inexperienced human. In practice

it outperformed our expectations, winning several games with humans. Fortunately for Splendor, the Greedy algorithm consistently loses to the other algorithms in this paper.

Chapter 4

Brute-Force Algorithms

4.1 Minimax

Brute force, or exhaustive search, is not so much a strategy as the lack of a strategy. A brute force algorithm tries to simulate every choice, and every choice after that, until the problem is solved. This technique is intractable for complex games; for example, it's estimated that the complete game tree for chess is over 10^{120} moves [1]. For games with partial information brute force is literally impossible, since the result of a turn is not available to the player. However, it is possible to simulate a few turns and use scoring heuristics to find a board state predictive of future victory.

Minimax is a well-known brute force algorithm that selects the maximum *guaranteed* value by assuming its opponent is running the same algorithm. The algorithm recursively generates boards for all legal moves for a specified number of turns. The boards generated on the last turn are scored. Then Minimax alternates between picking the move that led to the least favorable score and the move that led to the most favorable score.

Minimax has two weaknesses. One is computation speed. For each board there are 15 ways to take gems, fifteen cards that might be bought, and twelve cards that might be reserved. Many of these moves are illegal on any given board, but in general there are about twenty to twenty-five[1] legal moves every turn. This means that to simulate n turns, Minimax must generate 20^n boards. This delay is multiplied by the 50 or so turns in a typical Splendor game. On a typical personal computer[2] a three-turn simulation takes seconds, a four-turn simulation takes a few minutes, and a five-turn simulation takes hours. Since several hundred games are required to be statistically significant, most tests were run with Minimax(3).[3]

Minimax cannot possibly reach the game-end condition, as this would take orders of magnitude longer than the age of the universe. Instead, a heuristic must be used

[1]Interestingly, the average number of legal moves is heavily dependent on the algorithm and heuristics used.

[2]Program was run on a laptop with 8 gigabytes of memory and a 2.3GHz Intel i7-3610QM CPU.

[3]Minimax(n) refers to Minimax with a maximum search depth of n turns. For example, Minimax(3) looks at all legal moves on its turn, the opponent's turn, and then its turn once more.

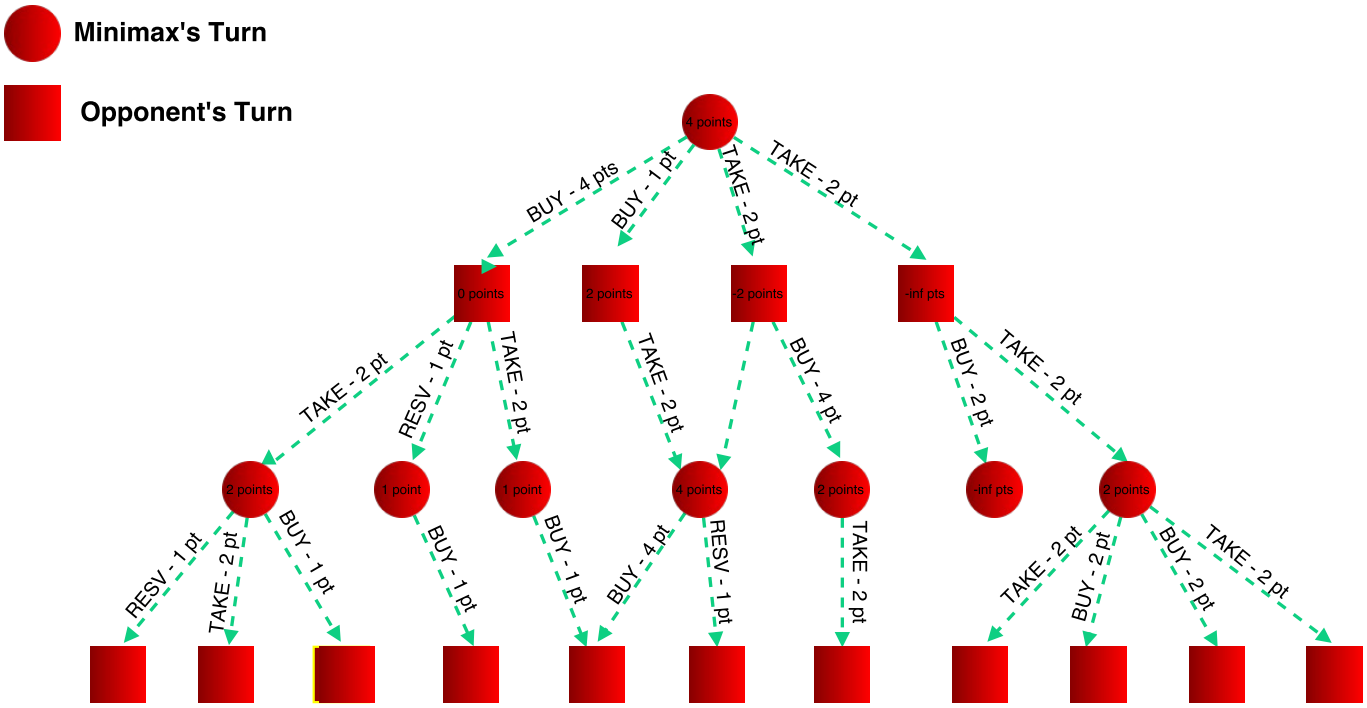

Figure 4.1: Possible game tree for Minimax. Moves from Minimax's turn are given a positive score, and moves from the opponent's turn are given a negative score.

to evaluate a non-terminal board state. We used the same heuristic as Greedy: A weighted sum of points, progress toward nobles, prestige, gems, and the win/loss condition.

The other weakness is partial information. Partial information didn't affect Greedy because decisions were only made in the turn with full information. Minimax on other hand is unable to predict which cards will be drawn after it buys or reserves a card. As the depth of the simulation increases, the game tree becomes increasingly inaccurate and holes are left on the board.

Despite these weaknesses, Minimax is significantly stronger than Greedy. Greedy and Minimax were tested over sets of games.

Minimax Depth	Minimax Wins	Greedy Wins	Minimax Win Rate
1	241	258	48 ±2%
2	356	141	72 ±2%
3	396	104	79 ±2%
4	16090	3910	80 ±0.2%
5	1655	345	83 ±0.9%

Table 4.1: Outcome of Minimax vs Greedy with the "allEval" fitness function. The last two entries were run on Frachtenberg's implementation owing to time constraints. The success of Minimax seems to improve quickly until depth 3, when it tapers off.

Given the same scoring heuristic, Minimax(1) ties with Greedy. In fact, at a depth of 1 Minimax is identical to Greedy. Both algorithms simulate all legal moves for the turn and choose the move which scores the highest. The performance of Minimax rapidly improves until depth 3, at which point additional depth gives marginal results.

An interesting observation was that the Buy:Reserve ratio steadily decreases over depths of Minimax. Minimax(1) has a Buy rate of 0.59 and a Reserve rate of 0.03. Minimax(5) has a Buy rate of 0.48 and a Reserve rate of 0.16. Take3 and Take2 rates held relatively constant around 0.35 and 0.02. This suggests that a good strategy is defensive play using Reserves to deny cards.

One hypothesis for the diminishing returns is that Splendor's game structure makes "Take" moves more or less equivalent. Depth 2 allows Minimax to predict which buys or reserves its opponent might choose, and Depth 3 allows Minimax to anticipate how those choices might affect it. Since players can rarely buy two cards in a row, Depth 4 doesn't add much information– one of the opponent's two simulated turns is likely to be a Take, which is difficult to oppose.

Another hypothesis is that the small element of chance makes it difficult to secure wins 100% of the time, even against a weak player. Therefore games between highly disparate players do not accurately measure player strength.

For this reason the Minimax trials were rerun against a stronger Minimax(3) player. In these trials the improvement per unit depth was roughly constant, invalidating the first hypothesis and suggesting that the second hypothesis is correct.

Minimax(n) depth	Minimax(n)	Minimax(3)	Minimax(n) Win Rate
1	2390	7610	23.9 ±0.4%
2	348	652	34.8 ±2%
3	455	445	45.5 ±2%
4	607	393	60.7 ±2%
5	738	262	73.2 ±2%

Table 4.2: Outcome of Minimax at various depths vs Minimax(3) with the "allEval2" fitness function. These entries were run on Frachtenberg's implementation.

The success of Minimax proves that the greedy strategy is not optimal. The positive correlation between Minimax depth and win rate suggests that even better strategies exist. It is likely that increasing the game tree depth would make Minimax more competitive, but the algorithm is too inefficient. We therefore explore randomized search strategies that are limited to a subset of the entire search space, but can look deeper into the game tree than Minimax.

4.2 Random Search

Random search is the process of moving down the game tree by choosing nodes at random. The Splendor.RandomSearch algorithm takes a size k and scoring function f as parameters. It then simulates a few turns into the future (optimized by binary

search at 10), choosing a random move for its own turns and using the Greedy algorithm to predict the opponent's turns. This process is repeated k times, yielding k final board states. Each final state is scored using f, and the move that led to the best state is chosen. Random Search converges to Minimax of the same depth as $k \to \infty$, since it will eventually simulate every move. In practice, the algorithm is much different since the game tree can be explored to a greater depth at the cost of exponentially-decreasing coverage[4].

Whether this is a good trade-off depends on the game. If good moves are rare, RandomSearch doesn't stand a chance– it will usually fail to evaluate those moves. If heuristics are bad, Minimax might be in trouble– its evaluation function will not be able to tell which end state is better at a shallow depth. Unfortunately, RandomSearch also uses heuristics– even if it simulated an arbitrary depth, it would run out of revealed cards and fail to end the game. Therefore, RandomSearch has an advantage when good moves are common, but the payoff from a good move is not apparent until many turns later.

Suppose Minimax (depth n) and RandomSearch (depth m) simulate the same number of boards. In other words, they are given equivalent computational resources. Assume $m > n$.[5] Minimax simulates the entire game tree up to its depth, approximately 25^n boards. RandomSearch simulates the same number, but the tree contains 25^m boards. Then RandomSearch only evaluates 25^{n-m} of the possible board states, or 4% if its depth is one greater than Minimax's.

The results of RandomSearch against each opponent under fixed parameters is shown in Table 4.2. The results are not easily comparable since opponents use differing computational resources, but they allow an estimate of RandomSearch's strength.

Opponent	**R.S. wins**	**Opponent wins**	**Percentage**
Greedy	403	55	88 ± 2%
Minimax(3)	51	49	51 ± 5%
ExactMove (depth 10)	78	22	78 ± 4%
BuyOrder (depth 10)	52	48	52 ± 5%

Table 4.3: Outcome of RandomSearch (Depth 10, 1000 searches) vs various algorithms with the "allEval2" fitness function. All algorithms except Greedy have similar computation time.

[4]for constant computational resources

[5]It doesn't make sense to use RandomSearch with $m \leq n$. RandomSearch is likely to evaluate some boards multiple times, so it is slower than Minimax and can still miss moves.

It should be noted that our implementation of RandomSearch is not completely random. We used the deterministic Greedy algorithm to simulate the opponent's moves, which decreases the size of the game tree by a factor of $25^{depth/2}$. When the opponent is actually using the Greedy algorithm, this gives RandomSearch clairvoyance; unless an affordable card is drawn, the opponent will take the move that RandomSearch predicted.

The same design choice was made for the three genetic algorithms in the next section. The Greedy algorithm is a good training opponent because it performs well for its computation time. Since Greedy was used as the primary benchmark, over-fitting to the opponent could not possibly make the algorithms worse. Had an algorithm performed exceptionally well versus Greedy but poorly against other algorithms, the next step would have been to dynamically generate simulation opponents.

Figure 4.2 shows the spread of search results over a number of rounds. This figure will serve as a useful comparison with the genetic algorithms in the following chapters. It should be noted that GAs require fitness to be nonnegative, so the range of results is more similar than it appears.

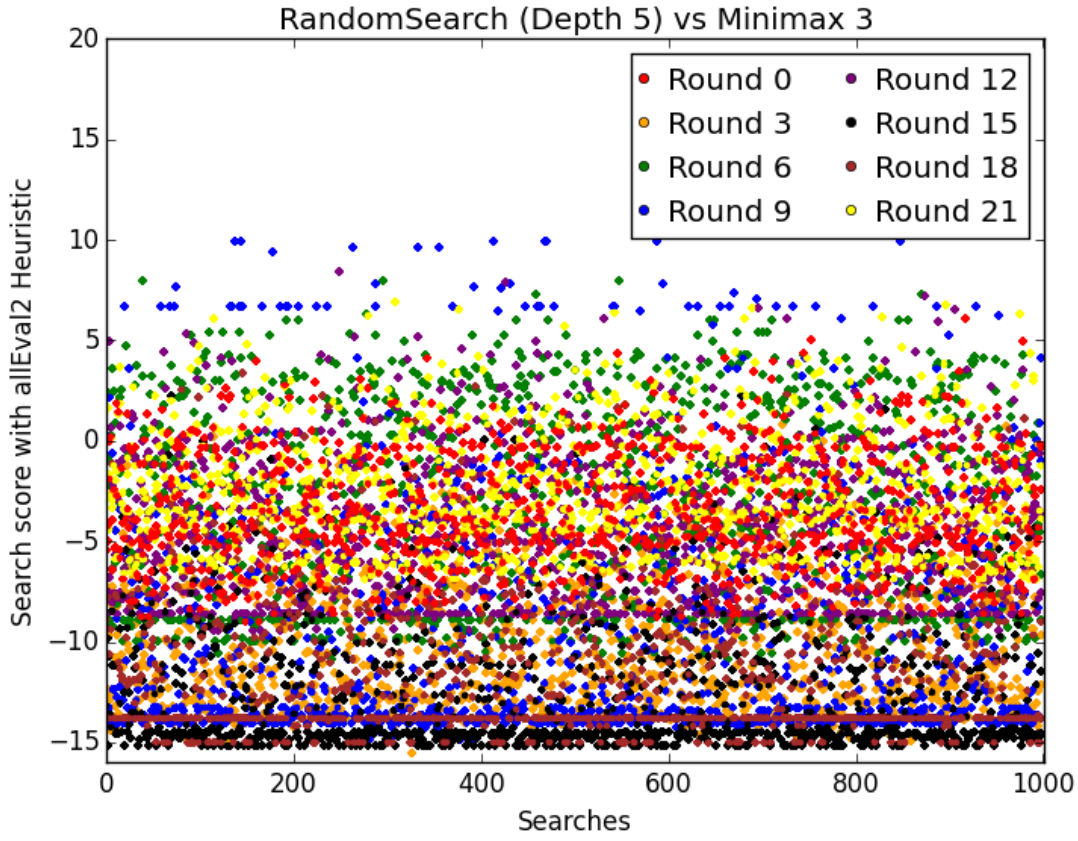

Figure 4.2: Early to mid-game search scores for RandomSearch vs Minimax(3). This game ran for 32 rounds in total, but later rounds are not displayed because the high-value WinLoss condition distorts the plot. Values for each turn are widely distributed, indicating a comprehensive search domain.

Chapter 5

Genetic Algorithms

5.1 What Are Genetic Algorithms?

Genetic algorithms are an approach to optimization that can be thought of as searching for building blocks to a solution. Like random search, a genetic algorithm begins by following a number of paths down the game tree and scoring the results with a heuristic. However, instead of committing to the highest scoring path, the algorithm attempts to recombine components from several paths to generate an even better result.

Genetic algorithms were contrived by analogy with biological evolution, and computer scientists continue to use the same terminology whenever possible. A genetic algorithm works in the following way:

1. Generate a number of randomized solutions, called "chromosomes".
2. Chromosomes are evaluated according to a fitness function.
3. Those with low fitness are pruned, and the remainder are duplicated.
4. Operators such as mutation and recombination are applied to adjust the chromosomes.
5. Repeat steps 2-4 a number of times. Each repeat is known as a "generation".

[14]

Number 4 is the defining step. Since the representation of chromosomes and operators are specific to the problem, the "traveling salesperson problem" (TSP) will be used as an example. TSP presents a map of cities with distances given between each city and asks for the shortest path that visits each city once.

Chromosomes for this problem are lists of cities. Each chromosome contains every city exactly once, in some random order. The fitness function consists of drawing a path from city to city in the order described by the chromosome and computing the length of the path. Two operators will be used: Mutation and Crossover. Mutation is a unary operation that takes a chromosome scrambles the list of cities; in this case,

two cities on the list will be swapped. Crossover is a binary operation that takes two chromosomes and returns the first with its list partially reordered according to the second.

By repeatedly pruning the population and applying the crossover operation, the algorithm is able to stitch together optimal subpaths. The mutation operator searches nearby solutions and periodically injects diversity so the population does not become homogeneous.

TSP chromosomes are an example of a permutation chromosome, which will be used later in this thesis.

When Should We Use Them?

Like greedy algorithms, genetic algorithms have some limitations on their use. The crucial piece of a GA compared to other search techniques is the crossover operation; otherwise the algorithm is just an iterated random search. Crossover is only effective if the solution space can be composed from sub-solutions For example, in the Traveling Salesman problem the shortest path among a dozen cities is likely to be contained within the shortest path among all of the cities, and a longer contiguous path cannot be optimal. This is analogous to the optimal substructure quality required by greedy algorithms.

5.2 IndexMove

The first attempt at a genetic algorithm was IndexMove. IndexMove represents Splendor strategy as a list of moves encoded by their index in the list of all currently-legal moves. The list of legal moves is deterministically generated, so a given chromosome always has the same meaning for a given board. For example, an IndexMove chromosome might contains the sequence $\langle 3, 14, 1, 5, 8, 12, 2 \rangle$. This would be interpreted as "Take the 3rd legal move on the board. Then take the 14th legal move on the resulting board. Then take the 1st legal move on the board that results from that." And so on. Since not all board states have the same number of legal moves, IndexMove takes move indices mod the number of legal moves. IndexMove is able to represent any possible strategy since the range includes all legal moves.

However, there are weaknesses with this encoding. IndexMove genes were contextual; The expression of a gene depended on the board state during evaluation, which in turn depended both on the gene's location in the chromosome and which genes preceded it. This meant that IndexMove chromosomes were unstable under crossover; moving a gene to a different chromosome changed the expression of that gene. IndexMove chromosomes were also unstable under mutation; changing a gene would affect the meaning of every gene thereafter. In biological terms, every mutation and crossover event was a frameshift mutation.

The instability of IndexMove chromosomes under genetic operations led us to quickly rule them out.

5.3 ExactMove

ExactMove was a close cousin to IndexMove. As the name implies, ExactMove chromosomes encode literal moves rather than indices of moves. An ExactMove chromosome looks like this: {⟨Take red,blue,green⟩, ⟨Reserve 21⟩, ⟨Buy 14⟩, ...}. Buys and Reserves are only generated for cards visible on the board at runtime.

ExactMove trades some of the weaknesses of IndexMove for others. Genes are stable under mutation and crossover; a gene refers to the same move wherever it is. Chromosomes, on the other hand, are not stable. The vast majority of chromosomes are illegal, containing at least one move which is illegal during simulation. Mutation and crossover early in a chromosome might lead to entirely different board states which invalidate other genes.

Several variations on ExactMove dealt with the unstable chromosome problem in different ways. One way was to skip illegal genes. However, this led to a very high proportion of junk DNA, genes which are carried around but never used.

Another way was to regenerate the tail end of chromosomes whenever an illegal gene was discovered. The regeneration was done during evaluation, after the list of legal moves had been generated for the simulated board. This way each new gene could be guaranteed legal. However, regenerating chromosomes used a large amount of CPU time.

In order to reduce the number of illegal chromosomes, crossover was restricted to genes with very similar board states. This was done by hashing each board state during simulation and associating the hash with the relevant gene. Crossover at the same index in both chromosomes was not allowed, as this would generate children identical to their parents. Despite this restriction, crossover still tended to wipe out chromosomes by invalidating genes after the crossover.

The Wargus team [8] encountered a similar difficulty evolving strategies for Wargus. They were able to divide the game into states based on available actions. Crossover was performed between identical states, guaranteeing that the daughter chromosomes contained legal moves. Unfortunately, there is no obvious way to categorize board states in Splendor.

Analyzing ExactMove

The best fitness in the population was recorded each generation, and plotted turn-by-turn as an indication of how well ExactMove evolved. The results were flat, indicating no improvement over generations. Frequently the fitness jumped significantly between turns, indicating a failure to predict the outcome of a turn. If turns were predicted with perfect clairvoyance and the GA always found the optimal solution, the fitness should only differ by the difference between the last-simulated turn and the first turn.

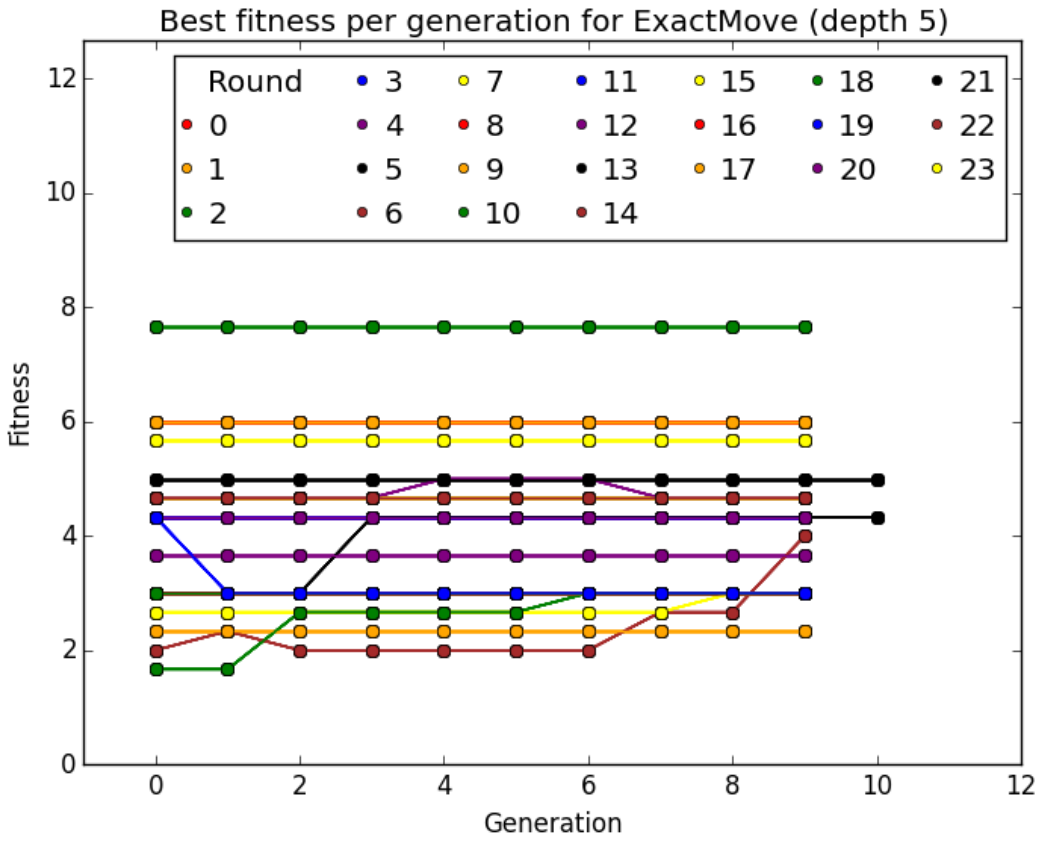

Figure 5.1: Best fitness per generation for ExactMove. Most turns are flat lines, showing no sign of evolution. However, the upward trend across turns indicates the potential for evolution.

ExactMove was evaluated without crossover and performed just as well, demonstrating that crossover offers no advantages over mutation. ExactMove was then run with the same number of evaluations but only one generation, removing all genetic operators and making it equivalent to random search. The new model performed much better than ExactMove with operators, demonstrating that generating fresh chromosomes is more effective than hill-climbing with mutation. This suggests that the fitness landscape is highly irregular.

The inability to produce stable genetic operators for IndexMove and ExactMove led us to explore other options for encoding strategies.

5.4 BuyOrder

BuyOrder was designed to encode abstract strategies that were stable across different boards. BuyOrder chromosomes are permutation chromosomes: Each gene is an integer, and no genes are repeated in the same chromosome. Mutation is a swap of two genes, and crossover involves an overlay of the orderings of two chromosomes. Permutation chromosomes are commonly used for problems such as the Traveling Salesman, where actions cannot be repeated and the ordering of genes matters.

The values of the genes in BuyOrder chromosomes are mapped to cards visible on the board when BuyOrder runs. For example, if there are twelve cards on the board and two in reserve, then a BuyOrder chromosome might be ⟨1,4,0,2,5,8,11,13,6,3,12,7,9⟩. As usual, the fitness function simulates several turns into the future. Whenever the BuyOrder player needs to make a move, it runs the greedy sub-algorithm "BuySeeker" on the next available card on the list.

BuySeeker follows a short deterministic protocol. Given a card as input:

1. If the card is affordable, Buy it this turn.
2. Else if the opponent can buy the card, Reserve it this turn.
3. Otherwise, iterate through all legal Takes. Choose the one that progresses the most towards the card. In case of ties, prioritize Take3 over Take2.

BuyOrder might seem inefficient at first glance. Instead of generating one new board state per gene, it must run an algorithm that explores every legal Take. In many cases that's ten board states per simulated turn. BuyOrder generates about ten times as many boards as ExactMove for the same search depth.

However, BuyOrder is still vastly more efficient that Minimax. The differences between the search techniques can be thought of this way:

- Greedy doesn't search any paths, but examines its own possible moves and chooses one.
- Minimax searches the entire game tree, an exponential number of possible moves.
- True RandomSearch searches individual paths, fixing all moves. Our implementation runs the Greedy algorithm on the opponent's turn, making it more like ExactMove.
- ExactMove and IndexMove fix their moves and examine the opponent's possible moves. This allows them to always take the correct branch for the opponent's turn (against Greedy).
- BuyOrder examines all possible moves each turn. Unlike Minimax, it fixates on one move before moving down the game tree, avoiding exponential explosion of the search space.

Initial results for BuyOrder appeared no more promising than for ExactMove. The generational best-fitness plot (Figure 5.2) looked largely the same. The win rate against Greedy was slightly higher, but not competitive with Minimax. Analysis of the population provided one upshot. BuyOrder chromosomes were much more stable than ExactMove chromosomes. Within each generation, the first five genes were almost always identical or exchanged by a (12) permutation.

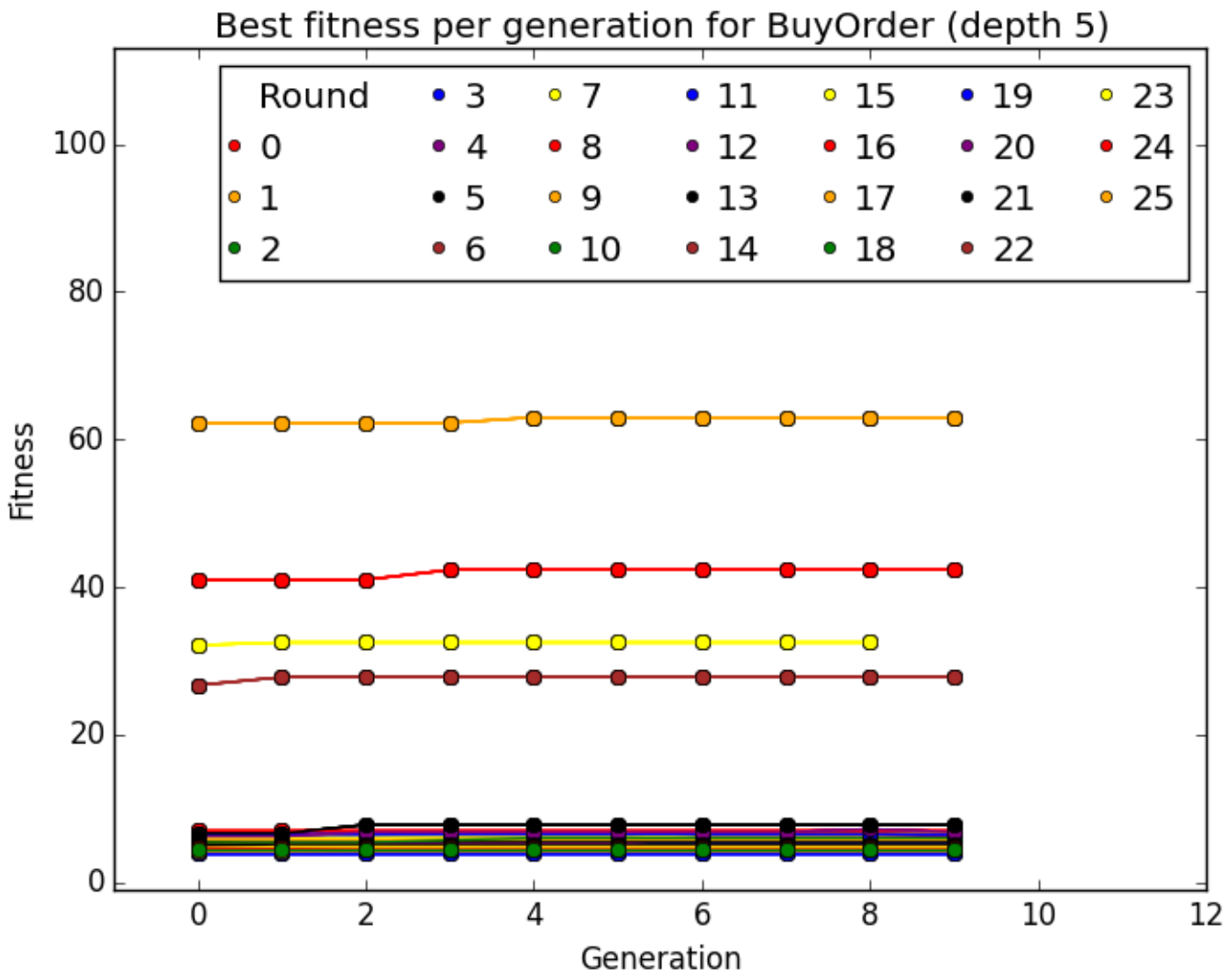

Figure 5.2: Best fitness per generation over one full game for BuyOrder vs Greedy. Each color represents a turn. As with ExactMove, fitness increases between turns but not between generations.

Genetic algorithms fail when the fitness function does not adequately represent success, or when the chromosome representation does not demonstrate optimal substructure. The success of Minimax and Greedy suggests that the allEval heuristic is suitable for predicting victory in Splendor. Does Figure 5.2 corroborate this view? Fitness increases as victory nears, but for the majority of the game fitness remains static. This indicates that the heuristic is not very predictive until the last few turns. The problem might not be with heuristic, but with Splendor itself. It is possible that most turns don't have a strong effect on the game's outcome.

If BuyOrder were evolving, we would expect fitness to increase with generations until it was comparable to the next turn's starting fitness. This clearly doesn't happen

in the last five turns, but it also can't happen earlier because fitness is constant across multiple turns.

The problem is even more apparent in Figure 5.3. At a glance it seems the population is too homogeneous; even at higher fitness, nearly all chromosomes have the same value. It turns out the problem is not with the population. On average, the first five buys of a BuyOrder chromosome are evaluated.[1] We therefore regard chromosomes with the first five buys identical as equivalent under evaluation. The number of chromosome equivalence classes was tallied at the end of each turn.

[1]Five buys might take ten or more turns, at which point nondeterminism adversely affects the tree search.

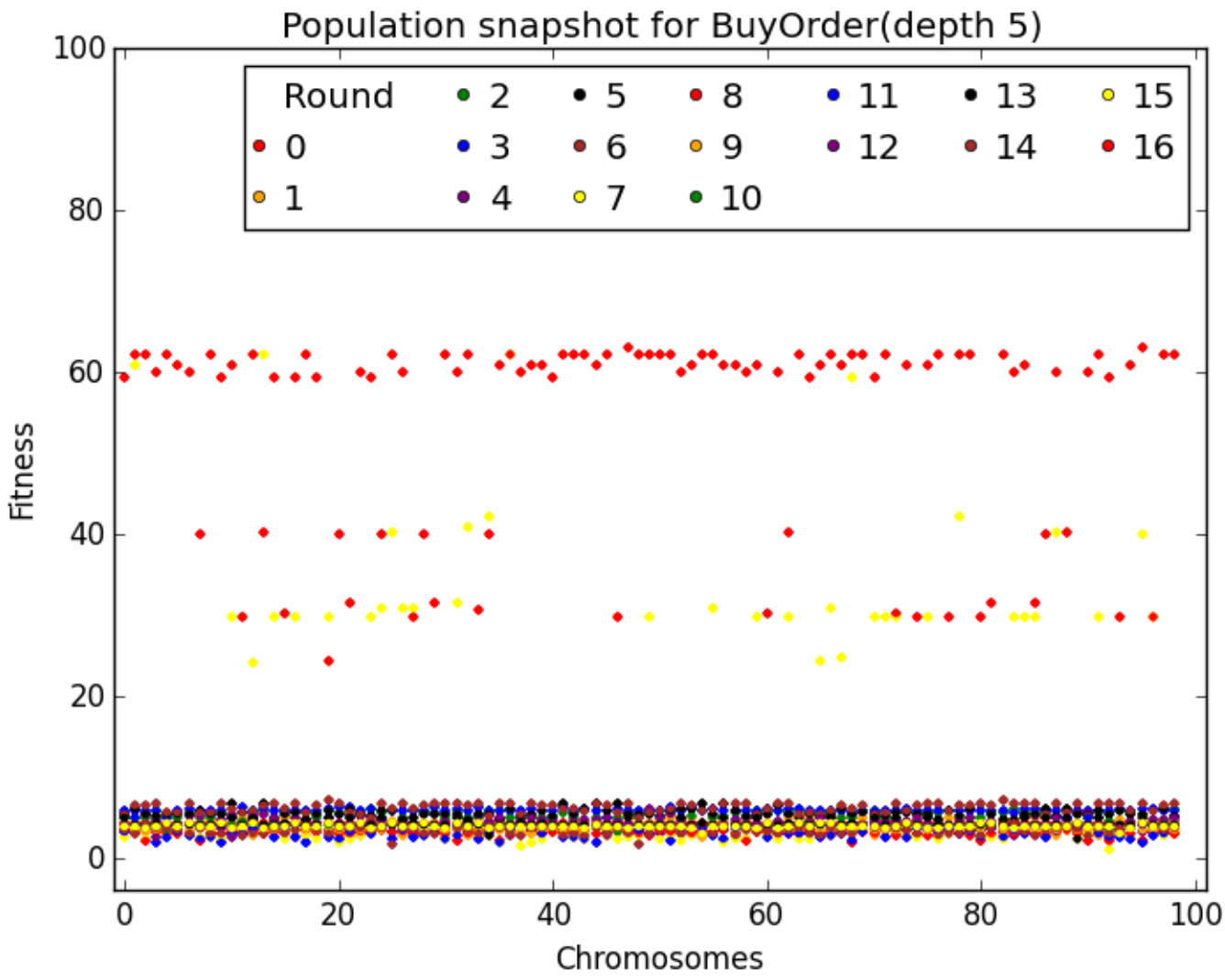

Figure 5.3: Population snapshots for a game between BuyOrder and Greedy. Each color represents a generation, and every fourth generation is represented. Most chromosomes in a given turn have the same fitness, indicating that the population is not very diverse compared to RandomSearch in Figure 4.2.

We learned from ExactMove that crossover could be problematic. The purpose of the crossover operator is to converge toward a solution that preserves the best parts of each sub-solution, but poorly-implemented crossover destroys the sub-solutions and produces a strategy that is essentially widespread mutation. This "destructive crossover" is likely to be even less effective than point mutation. Suppose 10% of point mutations improve fitness, and destructive crossover causes the equivalent of four point mutations. Then the mutation operator has a 10% chance of improving fitness, while the crossover operator has a $0.1^4 = 0.01\%$ chance of improving fitness.

We ran tests with several BuyOrder variants (Table 5.1) to determine whether we

were seeing destructive crossover.

BuyOrder Variant	Population Size	Evaluations	Win Rate (±4%)
Mutation Only	100	1000	79%
Crossover Only	100	1000	82%
Restricted Mutation Only	100	1000	81%
Restricted Mutation	1000	1000	80%
Restricted Mutation	100	1500	82%

Table 5.1: BuyOrder Variants vs Greedy with "allEval" Fitness Function. Variants included mutation-only (no crossover); crossover-only; mutation-only with mutation restricted to the first few (expressed) genes; and crossover plus mutation on the first few genes.

Table 5.1 suggests that neither crossover nor mutation had much effect on win rate. This is consistent with the fitness plot in Figure 5.2. Application of genetic operators just doesn't affect population fitness. As additional evidence, Greedy was run against BuyOrder with 1,000 chromosomes and one generation. BuyOrder's win rate was 80% over one hundred games, identical to the win rate with 100 chromosomes and ten generations. This demonstrated conclusively that evolution was not contributing to the algorithm's performance.

The final piece of evidence was the crossover statistics. Table 5.2 shows the total number of crossovers throughout one game, and the frequency with which crossover produced a child with higher fitness. It is apparent that crossover is not very effective. In fact, it compares unfavorably with mutation.

It is hard to determine why crossover fails for BuyOrder. In ExactMove it was readily apparent that crossover produced illegal moves, invalidating sub-solutions from both parent chromosomes. BuyOrder chromosomes don't contain moves, but card priorities, so invalidation is impossible. However, crossover might disrupt strategies in a similar way.

BuyOrder chromosomes evaluate with the help of the BuySeeker sub-algorithm. BuySeeker abstracts away the process of saving up for a card, at the cost of denying BuyOrder the ability to fully control its moves. It is possible that optimal play is impossible for BuyOrder, because BuySeeker does not encode those moves. It is also possible that changing the order of target cards changes the move sequence output by BuySeeker in a way that disrupts the strategy.

The first possibility is impossible to address without knowing optimal play, which is a catch-22. A back of the envelope calculation[2] puts the size of the game tree (up to depth 10) at $20^{10} - 30^{10}$. Assuming a full board, there are $12! - 15!$ (depending on reserved cards) combinations of BuyOrder chromosomes. Then the ratio of games to chromosomes is $7 \times 10^0 - 1 \times 10^6$. In practice only the first few targets on each chromosome ever get evaluated, so the ratio is even higher: $\frac{30^{10}}{12!/7!} = 6 \times 10^9$. While

[2]There are generally twelve cards to reserve, five colors for Take2, ten combinations for Take3, and a few legal buys.

not conclusive, it is believable that many good strategies are lost outside the domain of BuyOrder.

It could be that crossover disrupts the move sequence for each target by changing the starting conditions when BuySeeker sees that target. But this would not necessarily constitute a destructive crossover. If strategy can be generalized to the order in which one buys cards, then changing the move sequence between buys should not have much effect.

Pop. Size	**Evaluations**	**Improvements / Crossovers**	**Percentage**
100	200	670 / 2221	30.2%
100	500	1120 / 4932	22.7%
100	1000	1326 / 11005	12.0%
100	10,000	1321 / 95438	1.4%
1000	10,000	13782 / 103361	13.3%

Table 5.2: Crossover efficiency in BuyOrder (depth 10) expressed as the number of crossovers which generate child chromosomes with higher fitness. Note that the last entry has a larger population for the same number of evaluations, meaning there are fewer generations.

Table 5.2 suggests that crossover is doing something useful. The percentage of beneficial crossovers decreases with evaluations, which is to be expected as the population converges. However, the table does not align with Figure 5.2 or 5.3, which showed a static population. Examination of the crossover data revealed that over 90% of crossover improvements happen in the first generation. The population then immediately converges.

In fact, over a full game the average ratio of chromosome equivalence classes to total chromosomes was just 6%. This is undesirable as 94% of processing time is spent rehashing old solutions. Normally rapid convergence implies that the crossover rate is too high, there are too many generations, or the initial population is not diverse enough. The usual solutions, expanding the population or reducing the crossover rate, failed to address the problem.

Depth	Population Size	Evaluations	Win Rate (± 2%)
2	500	1000	53%
2	500	5000	57%
4	100	5000	70%
4	300	5000	64%
6	100	3000	79%
6	300	3000	84%
10	100	2000	85%
10	300	2000	88%

Table 5.3: BuyOrder vs Greedy With Varying Search Depths

Chapter 6

Discussion

The difficulty with designing a GA for Splendor comes down to the disruptiveness of Splendor's state transitions. Buy actions are particularly disastrous because they modify nearly every component of the board state; the number of cards available to Buy or Reserve decrements, the player's purchasing power may shrink dramatically, and the number of available Take options increases for both players. This means that it's hard to predict future board states without simulation. Genetic algorithms require that solutions which contain pieces of the optimal solution will be better than solutions which don't. This requirement is violated if pieces of the optimal solution become illegal when separated from the optimal.

ExactMove fails on both counts. Moves are not interchangeable building blocks because they have state dependencies. Optimal substructure is not guaranteed; for example, a chromosome with all of the Takes from an optimal solution might perform badly if it contains the wrong Buys. It's also possible to reach identical board states via several different paths (for example, by exchanging the order of two moves). Therefore two chromosomes might score well and play "the same game" from a human's perspective, but a mixture would be completely inviable.

BuyOrder fails for different reasons. Its strategy is coherent over generations, at the cost of being oversimplified. The weakness of BuyOrder seems to stem from the reduced solution space rather than the genetic operators. The original population is just too homogeneous for the operators to have anything to work with. The working assumption behind BuyOrder was that strategy can be reduced to Buys because there are many sequences of Takes which yield equivalent results. This assumption is not always true because gems are limited, and one series of Takes might deny the opponent while a permutation allows the opponent to deny BuyOrder. But in general it appears to be rare for gem piles to run out.

More importantly, BuyOrder can't evolve tactics such as the order in which to take gems or when to Reserve to guarantee a Buy. For example, the helper algorithm assumes that the opponent will not make offensive Reserves (denies); this is probably a valid assumption against Greedy, but completely naive against Minimax or any algorithm with depth greater than 1. The author believes it is necessary to abstract away tactics in order to evolve strategies, but it may be possible to design a hybrid algorithm that evolves strategies and then fine-tunes the tactics.

BuyAndDeny

Analysis of Minimax revealed that deeper searches tend to focus more on denying cards from the opponent. BuyOrder can choose to deny a card by placing that card first in priority, and will even Reserve the card if necessary. Unfortunately, the algorithm is subsequently forced to buy the card, even if ideal play would be to reserve and move on.

In practice, this can nevertheless lead to successful play. Suppose that optimal play involves denying Card X with a Reserve and then buying Card Y. On turn A BuyOrder sees that its opponent benefits immensely from buying Card X; therefore the algorithm prioritizes it. In simulation the algorithm reserves Card X and then goes on to take the necessary gems and buy the card. On turn A+1, Card X is no longer important and BuyOrder prioritizes Card Y, which now offers the greatest advantage in simulation. The algorithm has successfully reserved and moved on, matching optimal play. However, this solution is an accident; it wasn't simulated on turn A and the fitness for turn A was lower than it could have been. It's easy to imagine cases where buying Card X means losing Card Y, and the entire solution is scrapped.

BuyOrder was modified very slightly to address this weakness. Instead of storing a list of Buy priorities, the new algorithm stores a list of Buy and Reserve priorities. This doubles the size of a chromosome and increases the solution space from 20! to 40!. The new algorithm, dubbed "BuyAndDeny", demonstrated evolution for the first time as shown in Figure 6.1.

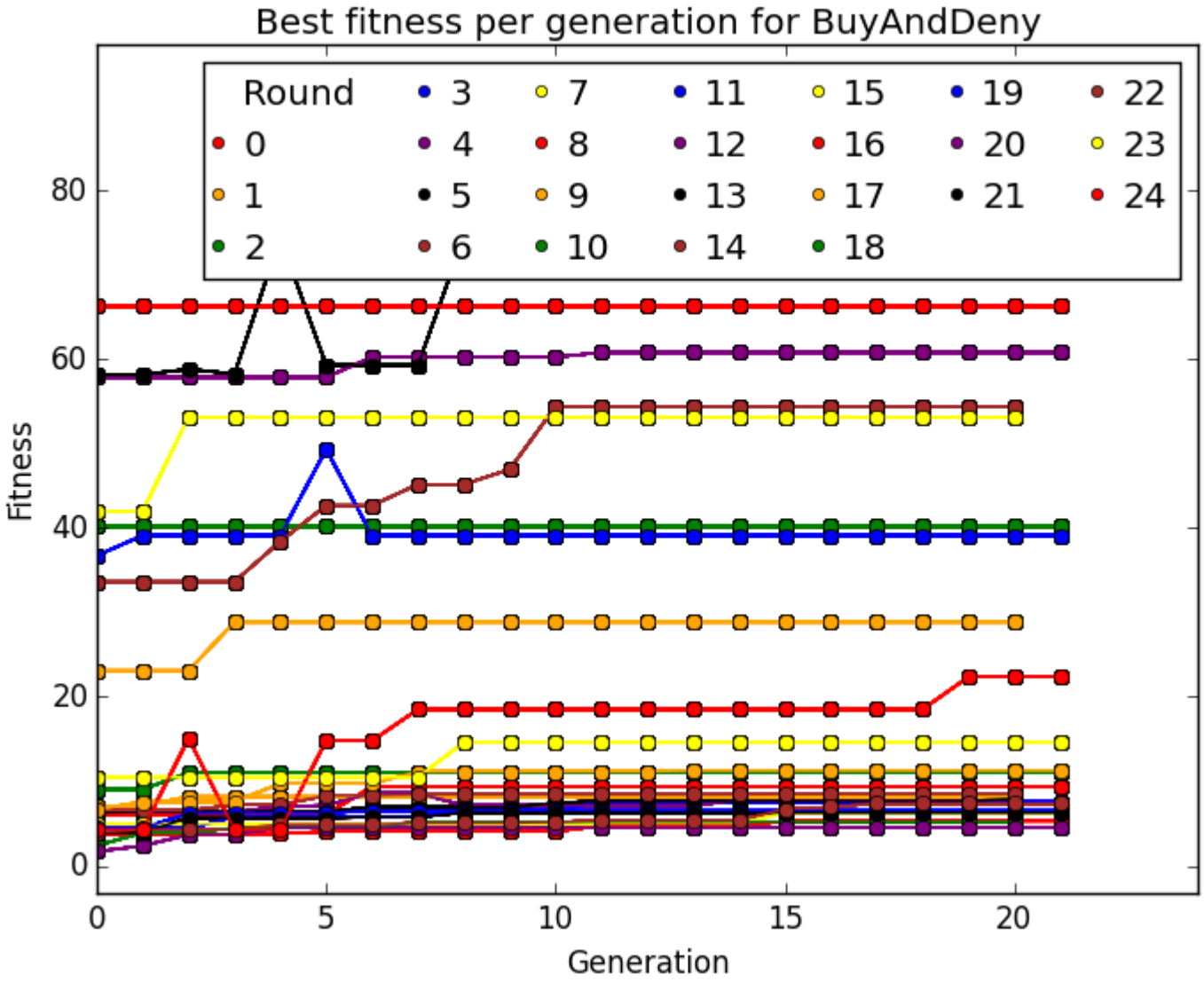

Figure 6.1: Best fitness per generation for BuyAndDeny (Depth 10). Several turns display minor improvement over a few generations, but generations 11-20 are largely wasted.

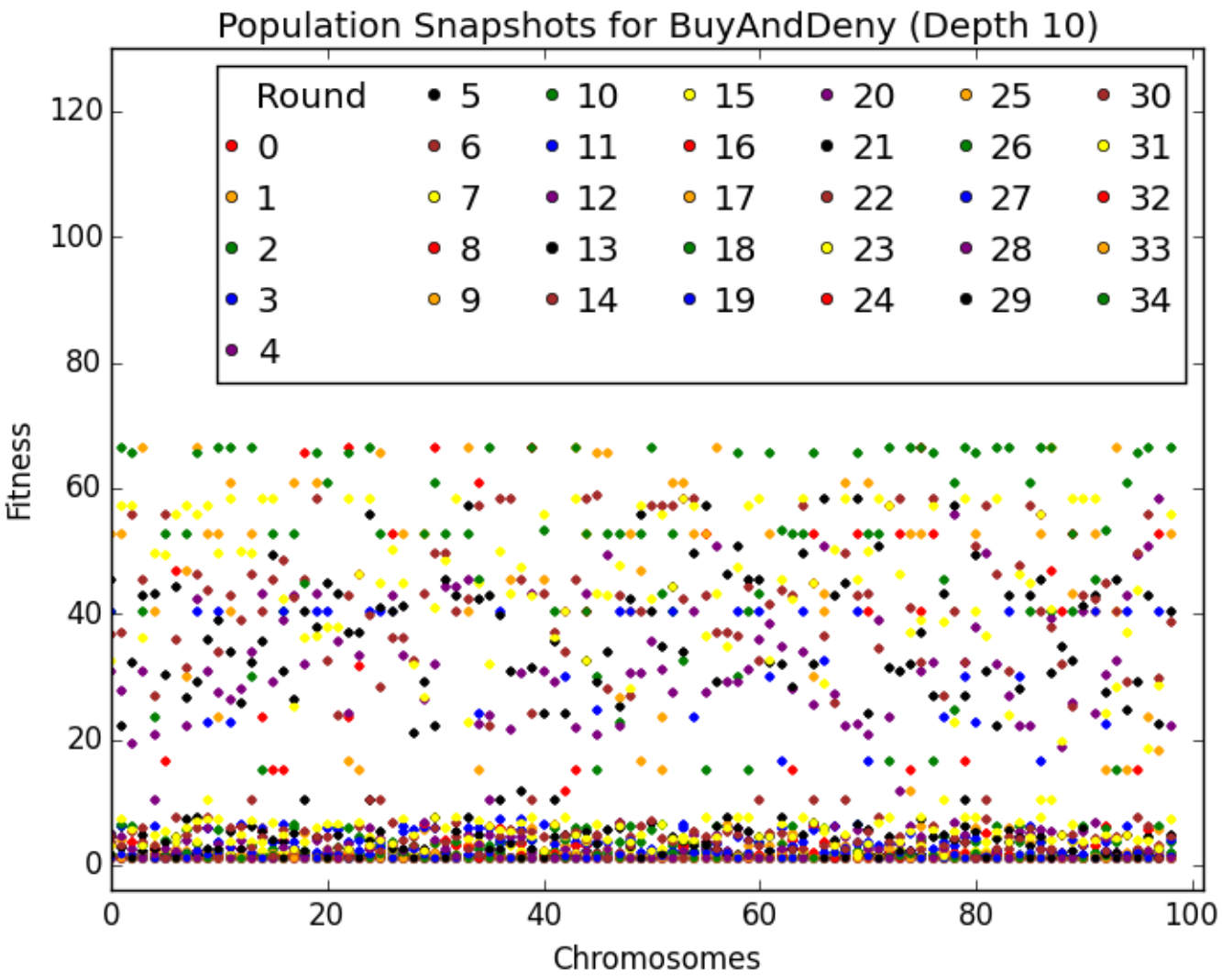

Figure 6.2: Population snapshots for BuyAndDeny (Depth 10). The population is significantly more diverse than in previous algorithms, but there is still clustering at lower fitness levels.

BuyAndDeny is still a far cry from a "successful genetic algorithm". The vast majority of generations yield no improvement to fitness. The algorithm could be optimized by reducing the number of generations and increasing the population size or search depth. However, it's unlikely that these parameter tweaks would generate evolution in turns that show no sign of it. An abundance of best-moves makes it more likely that BuyOrder will immediately find a solution with the best fitness and have no room to evolve. Further expansion of the search space might reveal better solutions, but would also decrease the probability of finding good solutions.

Analysis of Minimax (depth 5) revealed that on average, 2.5 first-moves yield the best fitness. In other words Minimax does not usually find a single best move, although it narrows the candidates by about 90%. Minimax (depth 3) found an average of 3.1 moves with the best fitness. The number of best-moves per turn varied from 1 to 10.

This statistic should not be surprising to a Splendor player. Often neighboring moves may be legally permuted, particularly if the permutation does not cause a Buy to come before a necessary Take. These permutations might easily yield the same fitness.[1] A more complex heuristic might differentiate permutations, but to some extent this indicates that many move order is not critical. That's good for BuyOrder, which abstracts away the order of its Takes.

Assume that some turns have many optimal solutions, i.e., many moves yield the best fitness. This makes it more likely that a GA will initialize with at least one optimal chromosome. Can this information be used to improve BuyOrder or the other algorithms? One area for optimization is time constraints. Competitive games like chess often have cumulative time constraints; players are allotted time per game. An algorithm can exploit unequal turn length by terminating early on turns which seem easily solvable. The extra time can later be used to explore deeper in the game tree on turns that matter.

[1] Depending on the heuristic used. At the time this data was taken, the heuristic included exponential decay in simulation depth. The decay function differentiates permutations on two or more moves with differing fitness.

Conclusion

Six tree-searching algorithms (Greedy, Minimax, RandomSearch, IndexMove, ExactMove, BuyOrder) were compared for a clone of Marc Andre's Splendor. The Greedy algorithm performed surprisingly well, playing competitive games against humans. Minimax dominated Greedy even at a depth of 2, but the win rate tapered off by depth 4. Games between Minimax variants indicated that performance is linear with search depth until an 80-90% win rate is achieved.

RandomSearch with a depth of 10 turns performed about as well as Minimax with a depth of 3 turns when given similar computational resources. This indicates that Splendor's modest nondeterminism is not a limiting factor, even when no attempt is made to simulate card draws.

IndexMove and ExactMove depict the player's strategy as a list of moves. The representation proved to fit the game structure poorly because moves in Splendor depend on previous moves. Therefore any operator is likely to invalidate the entire strategy. This flaw was alleviated by restricting crossover to similar board states; however, the solution had the undesirable side effect of limiting crossover to move sequences with little impact on game state.

BuyOrder represents strategy as a priority of buys and denies. Without denies, BuyOrder performed slightly worse than Minimax under similar resource constraints. The genetic operators were stable; however, the search space was too narrow and moves which may have been optimal were inaccessible. When BuyOrder was expanded to include denies (reserves), the algorithm performed better than Minimax(4) against Greedy but continued to lose a majority of games to both Minimax(3) and Minimax(4).

Many board games have in common with Splendor highly state-dependent moves. It is likely that GAs based on fixed moves will not work well on any such games. The results from BuyOrder suggest that a promising alternative is a hybrid GA which encodes moves, but allows for more flexible strategies. For example, the move <Take red,blue,green> could be substituted with <Take red,black,green> if no blue gems are available.

Future Work

Although we were unable to find a GA representation with effective crossover, there is no evidence that Splendor is impenetrable to genetic algorithms. The space of possible algorithms is endless, and several alternative representations emerged as a result of this work. The success of RandomSearch demonstrates that deep searches are comparable to wide searches, although neither RandomSearch nor Minimax were fully optimized.

One possibility is modifying ExactMove so that groups of related moves are chunked together. Intuition suggests that moves leading up to a buy are directed "towards" that buy, and can be grouped with it. If true, then crossover should affect an entire chunk rather than two or three moves.

The addition of Reserves improved BuyOrder, and it's reasonable to think that further improvements could be made. Whether the algorithm can perform significantly better than Minimax is an open question. The following are some possibilities for future investigation:

Coevolution - BuyOrder and the other GAs simulate an opponent using a Greedy algorithm. This presumably means that BuyOrder overperforms the Greedy player and underperforms Minimax. One way to simulate better opponents is to evolve an opponent in parallel, pitting random members of the two populations against each other to determine fitness. This two-space evolution is utilized to good effect in [13].

Expanded search space - ExactMove explores the full solution space, but its genes are too fragile. BuyOrder genes are stable because they express desired goals and allow the sub-algorithm to meet those goals. These algorithms could be combined by converting exact moves into preferences. For example, "TAKE red,blue,green" could be as interpreted as "choose the move that is closest to TAKE red,blue,green" or "gain a red, blue, and green gem (in as few turns as possible)". The resulting search space includes all legal moves, and illegal moves are converted into the closest legal approximation.

Hybridization Hybrid algorithms are a well-known approach for making GAs compete with established search algorithms. [14] One easily applicable hybridization technique is the use of an established search algorithm to generate the initial population. In our case, Minimax or RandomSearch would be used to seed the population. The GA would then recombine RandomSearch solutions, or differentiate equivalent Minimax solutions by extending the search depth.

Splendor is nondeterministic over many turns, as each Buy or Reserve reveals a

new card. The implementation in this work made no attempt to replace cards, causing the board to become sparse by the end of deep searches. It may be worthwhile instead to simulate random draws.

A game with a higher branching ratio might be intractable to Minimax, giving other search algorithms an advantage. Splendor has a branching ratio of 20-25; for comparison, Chess and Go have branching ratios of 30 and 300 respectively. [2] However, Justesen et al. reported success using GAs for wide, shallow searches in a war game with branching ratios in the millions.

Appendix A

List of Splendor Cards

Table A.1: Nobles. Each noble is worth 3 points, so they are differentiated only by their color requirements (the types of cards required to achieve them). Each noble requires nine cards in three colors or eight cards in two colors.

Noble ID	White	Blue	Green	Red	Black
0			3	3	3
1	3	3	3		
2	3	3			3
3		3	3	3	
4	3			3	3
5	4	4			
6		4	4		
7			4	4	
8	4				4
9				4	4

Table A.2: Prestige Cards - Tier 1

Gem color	Points	White	Blue	Green	Red	Black
black		1	1	1	1	
black		1	2	1	1	
black		2	2		1	
black				1	3	1
black				2	1	
black		2		2		
black				3		
black	1		4			
blue		1		1	1	1
blue		1		1	2	1
blue		1		2	2	
blue			1	3	1	
blue		1				2
blue				2		2
blue						3
blue	1				4	
white			1	1	1	1
white			1	2	1	1
white			2	2		1
white		3	1			1
white					2	1
white			2			2
white			3			
white	1			4		
green		1	1		1	1
green		1	1		1	2
green			1		2	2
green		1	3	1		
green		2	1			
green			2		2	
green					3	
green	1					4
red		1	1	1		1
red		2	1	1		1
red		2		1		2
red		1			1	3
red			2	1		
red		2			2	
red		3				
red	1	4				

Table A.3: Prestige Cards - Tier 2

Gem color	Points	White	Blue	Green	Red	Black
black	1	3	2	2		
black	1	3		3		2
black	2		1	4	2	
black	2			5	3	
black	2	5				
black	3					6
blue	1		2	2	3	
blue	1		2	3		3
blue	2	5	3			
blue	2	2			1	4
blue	2		5			
blue	3		6			
white	1			3	2	2
white	1	2	3		3	
white	2			1	4	2
white	2				5	3
white	2				5	
white	3	6				
green	1	3		2	3	
green	1	2	3			2
green	2	4	2			1
green	2		5	3		
green	2			5		
green	3			6		
red	1	2			2	3
red	1		3		2	3
red	2	1	4	2		
red	2	3				5
red	2					5
red	3				6	

Table A.4: Prestige Cards - Tier 3

Gem color	Points	White	Blue	Green	Red	Black
black	3	3	3	5	3	
black	4				7	
black	4			3	6	3
black	5				7	3
blue	3	3		3	3	5
blue	4	7				
blue	4	6	3			3
blue	5	7	3			
white	3		3	3	5	3
white	4					7
white	4	3			3	6
white	5	3				7
green	3	5	3		3	3
green	4		7			
green	4	3	6	3		
green	5		7	3		
red	3	3	5	3		3
red	4			7		
red	4		3	6	3	
red	5			7	3	
white	3			3	3	3
white	3	3	3	3		
white	3	3	3			3
white	3		3	3	3	
white	3	3			3	3
white	3	4	4			
white	3		4	4		
white	3			4	4	
white	3	4				4
white	3				4	4

References

[1] C. E. Shannon, *Programming a computer for playing chess*. Springer, 1988.

[2] N. Justesen, T. Mahlmann, and J. Togelius, "Online evolution for multi-action adversarial games," in *Evostar 2016*. Springer, 2016.

[3] F. Hapgood, "Computer chess bad - human chess worse," *New Scientist*, p. 827, 12 1982.

[4] M. Campbella, A. J. Hoane Jr, and F.-h. Hsuc, "Deep blue," *Artificial Intelligence*, vol. 134, pp. 57–83, 2002.

[5] B. Brügmann, "Monte carlo go," Citeseer, Tech. Rep., 1993.

[6] I. Szita, G. Chaslot, and P. Spronck, "Monte-carlo tree search in settlers of catan," in *Advances in Computer Games*. Springer, 2009, pp. 21–32.

[7] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[8] P. Spronck and M. Ponsen, "Automatic generation of strategies," *AI Game Programming Wisdom*, vol. 4, pp. 659–670, 2008.

[9] T. E. Revello and R. McCartney, "Generating war game strategies using a genetic algorithm," *AI Magazine Volume 27 Number 3*, pp. 75–84, Fall 2006.

[10] S. J. Louis and C. Miles, "Playing to learn: case-injected genetic algorithms for learning to play computer games," *Evolutionary Computation, IEEE Transactions on*, vol. 9, no. 6, pp. 669–681, 2005.

[11] https://boardgamegeek.com/boardgame/148228/splendor. (2013) Splendor. [Online]. Available: https://boardgamegeek.com/boardgame/148228/splendor

[12] T. H. Cormen, *Introduction to Algorithms*, 3rd ed. Cambridge, Mass. : MIT Press, 2009.

[13] J. W. Herrmann, "A genetic algorithm for a minimax network design problem," in *Proceedings of the 1999 Congress on Evolutionary Computation*, 1999, pp. 1099–1103.

[14] L. Davis, *Handbook of Genetic Algorithms*. New York : Van Nostrand Reinhold, 1991.

[15] S. M. Lucas and G. Kendall, "Evolutionary computation and games," *IEEE Computational Intelligence Magazine*, vol. 1, no. 1, pp. 10–18, February 2006.

[16] S. J. Louis and C. Miles, "Playing to learn: case-injected genetic algorithms for learning to play computer games," *IEEE Transactions on Evolutionary Computation*, vol. 9, no. 6, pp. 669–681, Dec 2005.

[17] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, Mass. : Addison-Wesley Publishing Company, 1989.

[18] M. Andre, *Splendor*. Space Cowboy, 2014.

[19] M. Ponsen, H. Munoz-Avila, P. SPronck, and D. W. Aha, "Automatically generating game tactics through evolutionary learning," *AI Magazine Volume 27 Number 3*, pp. 75–84, Fall 2006.

View publication stats