The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-19)

Determining Solvability in the Birds of a Feather Card Game

Shuto Araki, Juan Pablo Arenas Uribe, Zach Wilkerson, Steven Bogaerts, Chad Byers
DePauw University
Greencastle, IN 46135, U.S.A.
{shutoaraki_2020, jarenasuribe_2021, zwilkerson_2020, stevenbogaerts, cbyers}@depauw.edu

Abstract

Birds of a Feather is a single-player card game in which cards are arranged in a grid. The player attempts to combine stacks of cards under certain rules, with the goal being to combine all cards into a single stack. This paper highlights several approaches for efficiently classifying whether a randomly-chosen state has a single-stack solution. These approaches use graph theory and machine learning concepts to prune a state's search space, resulting in significant reductions in runtime relative to a baseline search.

Introduction

In this paper we compare several search and machine-learning approaches to determining the solvability of a puzzle game, Birds of a Feather (BoaF) (Neller 2018). In this single-player game, cards are drawn randomly from a standard 52-card deck and placed face up in an r-by-c grid. Here, we focus on the $r = c = 4$ case. Each of the placed cards can be thought of as a stack of size 1. Under certain conditions, stack x can be moved on top of stack y, forming a single new stack at y's position with x on top, and leaving a blank space at x's former position. This is allowed only when x and y are in the same row or column and the stacks' top cards meet one of two conditions: 1) they share a suit, or 2) they have the same or adjacent rank, according to the ordering A, 2, 3, ..., J, Q, K, where A and K are not adjacent. The goal of the game is to combine stacks one move at a time using these rules until a single stack remains on the grid. If a given game state can be reduced to a single stack, then that game state is said to be *solvable*. Many (but not all) initial deals are solvable, but moves must be chosen carefully to avoid a mistake leading to an unsolvable mid-game state.

This research compares several algorithms for taking a game state (starting or mid-game) and determining whether or not that state is solvable. Some algorithms achieve 100% accuracy through search of a pruned game tree. Others avoid the game tree entirely, aiming for lower execution time, but also having imperfect accuracy. This paper describes each of these algorithms in detail, and concludes with an experimental comparison.

Copyright © 2019, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

A Baseline Approach: Constrained DFS

A solvable state has a game tree with at least one leaf representing a single-stack state. Thus one approach for determining solvability is to search for such a leaf. For early-game states, an exhaustive search is not feasible due to the size of the tree, but a simple pruning strategy serves as an effective baseline approach (Neller 2018). Specifically, once a solvable child is found, the parent may be declared solvable, with the other children pruned. Applying this pruning to depth-first search, we call this "Constrained DFS".

Dataset Generation

This work uses a randomly-generated dataset of 500,000 rows, each consisting of a game state, represented as a string, and a binary value indicating whether the state is solvable or not. The first random choice is the number of stacks, from 1 to 16, followed by the top card for each stack, and finally the row/column placement of each stack. Constrained DFS is then used to determine the solvability of the state. Additional input features are also generated, to be used by some of the algorithms. Below, two types of input features are described: graph features and card features.

Graph Features

To consider graph features, we first define two stacks to be *compatible* if the top cards are of the same suit, and/or have the same or adjacent ranks. Note in particular that compatibility does not consider the position (row and column) of the stack. We then define an undirected graph corresponding to a state, called a *compatibility graph*. This graph has one node for each stack, and an edge between nodes indicates that the corresponding pair of stacks is compatible. Such a graph is introduced in (Neller 2018).

Consider a state s with corresponding compatibility graph g. Suppose s has compatible stacks x_s and y_s. Therefore, g has an edge between the corresponding nodes x_g and y_g. If x_s and y_s happen also to be in the same row or column, then one can be placed on the other. For example, x_s might be placed on top of y_s. In that case, g is updated such that y_g and all edges connected to y_g are removed. If, on the other hand, x_s and y_s are not in the same row or column, the edge between x_g and y_g indicates the *potential* of the corresponding stacks being combined, if future gameplay results in a row or column match.

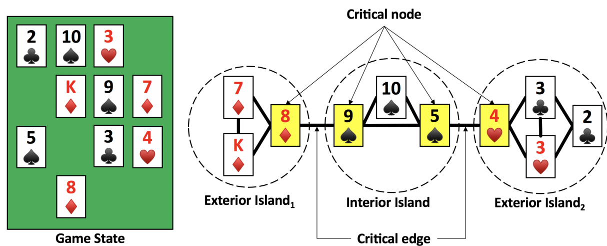

Figure 1: An illustration of compatibility graph definitions

If a compatibility graph g is disconnected, then the corresponding state is unsolvable. To see why, suppose g has two subgraphs g_1 and g_2 that are disconnected from each other. That is, there is no path between any node in g_1 and any node in g_2. Gameplay only removes nodes and edges in the compatibility graph; it never adds them. Therefore, no gameplay will create a path between any node in g_1 and any node in g_2. Thus, the nodes in these subgraphs are permanently incompatible, and so the state is not solvable.

On the other hand, a connected compatibility graph is not necessarily solvable, due to row and column requirements for valid moves. For example, consider a two-stack state with a 9♠ in the upper-right corner, and a 9♣ in the lower-left. The compatibility graph consists of two nodes with an edge between; it is connected. Due to the row/column requirement, however, the state is not solvable.

Figure 1 illustrates several terms for compatibility graphs. In graph theory, an articulation node is a node in a connected graph that, if removed, results in a disconnected graph. Thus, a BoaF state can become unsolvable by covering a stack corresponding to an articulation node in the compatibility graph. We define a *critical node* as a node in a compatibility graph that is either an articulation node or a node with exactly one connected edge. We also define a *critical edge* as an edge that connects two critical nodes.

Suppose g_1 and g_2 are subgraphs of g with no nodes in common, and that there is a single edge e connecting these two subgraphs. Let n_1 be the node in g_1 connected to e; similarly for n_2. Therefore, n_1 and n_2 are critical nodes and e is a critical edge in g. We define g_1 and g_2 as *islands*. That is, an island is a subgraph that is connected to the rest of the graph via only critical edges. If an island is connected to the rest of the graph via only one critical edge, it is called an *exterior island*; otherwise it is an *interior island*. Again, Figure 1 illustrates these terms.

Finally, we introduce the idea of *node connectivity*, defined as the minimum number of nodes that would have to be removed to disconnect the compatibility graph.

Given these ideas, the following compatibility graph input features are computed: minimum, maximum, median, and mean degree; critical node, critical edge, and island counts; is connected or not; and node connectivity.

Card Features

Several useful features of a given state can be generated via simple counts and descriptive statistics:

- RC count: For each stack, the number of other stacks in the same row or column (ignoring compatibility). Across the state, this can be expressed as a mean RC count, as well as a median, minimum, and maximum, for example.
- RC compatibility: For each stack, the number of legal moves available, considering both compatibility and position. This can be expressed as minimum, maximum, median, and mean values across a state.
- Ace count, Kings count: The number of aces and kings in the state. These are considered separately from the other cards, as they have fewer cards with which they are compatible by rank.
- Rank standard deviation: The standard deviation of ranks in the state, which corresponds somewhat with the likelihood that any two stacks are compatible by rank.
- Suit ratios: The ratio of the number of cards in a given suit to the number of remaining cards in the state, expressed as single values for each suit.

Algorithms

In this section, four novel solvability classifiers are described: Compatible DFS, Solvability Model, Ordered DFS, and Island Merge. Solvability Model is an approximation algorithm based on machine learning, while the others achieve 100% accuracy by searching pruned trees. We describe each approach here, with comparisons between algorithms in subsequent sections.

Compatible DFS

Compatible DFS is simply Constrained DFS (described above), with an additional optimization. Using the compatibility graph, Compatible DFS checks if the graph is disconnected. If so, then that subtree can be pruned, with the state determined unsolvable. This optimization maintains 100% accuracy, while also reducing the size of the game tree significantly. Below, the performance of this approach is compared experimentally with several others.

Solvability Model

The Solvability Model approach uses supervised learning to create a classifier, with the binary solvability value being the target output. We consider several classification algorithms.

Mean accuracy results are shown in Table 1. All models were tested via 10-fold cross-validation, as suggested in (Kohavi 1995), on a set of 500,000 randomly-generated states. For this initial experiment, the default hyperparameters of scikit-learn (Pedregosa et al. 2011) and LightGBM (Ke et al. 2017) were used for each model. Note that Table 1 also shows results of two baseline models, one always returning “solvable,” the other always “unsolvable.” Every other model’s accuracy surpasses these baselines. The most accurate model is LightGBM, achieving 94.44% accuracy.

LightGBM is an open-source machine learning algorithm implementation, with both classification and regression versions available. While a full discussion is beyond the scope of this paper, we provide a few key points. LightGBM uses gradient tree boosting, in which an ensemble of regression

Model	Accuracy
LightGBM (Ke et al. 2017)	0.94443
Bagging	0.94393
Multilayer Perceptron	0.94026
AdaBoost	0.93616
Random Forest	0.93579
Gradient Boosting Classifier	0.93300
Stochastic Gradient Descent	0.92962
Logistic Regression	0.92741
Decision Tree	0.91853
Always Unsolvable	0.51225
Always Solvable	0.48775

Table 1: Accuracy in the testing set for various models.

trees are built, with each tree hopefully correcting some error of the previous trees. One insight of LightGBM is in the order of tree construction. A best-first, rather than breadth-first, approach is used, in which the leaf with the highest prediction error is chosen for expansion first (Shi 2007). In addition, LightGBM uses histograms to discretize each continuous feature (Ranka and Singh 1998). The discrete bins of a histogram dictate the subsets into which the data is split on that feature in tree construction, and what branch is followed in application of the model. The use of histograms for discretization also allows smaller data types to be used, thus reducing memory usage (Li, Wu, and Burges 2008).

As LightGBM achieves the highest score of all of the tested models, it is chosen for hyperparameter tuning via a grid search. A grid search is an algorithm tuning process in which many models are created and tested. Each model is defined with a different combination of multiple hyperparameter values in an effort to find the ideal hyperparameters. Using 10-fold cross-validation to tune learning rate, number of estimators, and number of leaves simultaneously, the ideal combination is 0.2, 50, and 90, respectively. This tuned LightGBM model increases the mean accuracy to 94.479%, and thus is used as the canonical Solvability Model in algorithm comparison experiments below.

Unfortunately, at 94.479% accuracy, the tuned LightGBM model is still not accurate enough to be used reliably in solvability prediction. Furthermore, due to the complexity of some of the input features, this model is also fairly slow; in fact, for a state with less than six cards, it is typically faster to use a search algorithm. We revisit the use of a machine learning model in our discussion of Ordered DFS.

Ordered DFS

Returning now to search algorithms with 100% accuracy, note that both Constrained DFS and Compatible DFS search siblings in arbitrary order. If a solvable child is found, the remaining unvisited siblings are pruned. So more pruning occurs if a solvable child is found sooner. Thus, Ordered DFS places siblings in a priority queue, with higher priority assigned to children with a greater estimated likelihood of solvability. This is estimated using a regression model similar to the Solvability Model classifiers described above.

The challenge in Ordered DFS is to balance the time of execution of the regression model with the benefit of more precise estimates for sibling ordering. Two ideas can be considered to strike this balance. First, if the full set of features described above is too expensive, then perhaps some subset of features can provide a more effective balance. Second, ordering may be applied selectively. Such ideas may bring increased processing speed per node, but may also lead to increased ordering error, meaning less pruning. Thus a careful balance of tradeoffs is considered in the experiments below.

Input Feature Cost-Performance Analysis This first experiment considers the tradeoff that a greater number and complexity of input features leads to an improved order, but also longer ordering time. We refer to a particular experimental configuration as an Ordered DFS *system*. In this experiment, each system varies in the subset of features used, and is based on either LightGBM or linear regression. LightGBM is considered because it is the most accurate in the Solvability Model work above. Linear regression is considered for its great simplicity and speed, as a counterbalance to the comparative complexity of LightGBM.

Rather than explore every possible feature subset, subsets are chosen based on 1) relationships between feature calculation requirements, and 2) time of calculation. For example, once the expense of looping through a state is undertaken to, say, count the number of aces, the program might as well count the number of kings, as well as the suits (for suit ratios) and open spaces (for RC count values), among other card features. Similarly each graph feature requires the compatibility graph. If any one graph feature is included, then the remaining ones can be included at little additional cost. Exceptions to this grouping approach occur in some more expensive calculations, like island count. Finally, note that the number of stacks is not considered in any feature subset for Ordered DFS, since Ordered DFS always compares siblings, and siblings always have the same number of stacks.

Twelve subsets of features are tested here, as listed in Table 2. This leads to 24 Ordered DFS systems using either linear regression or LightGBM. In either case, the default scikit-learn hyperparameters are used. The model is trained on a $500{,}000$-element random dataset, using only the designated subset of features. The resulting model is used in Ordered DFS. Each such Ordered DFS system is tested on a separate randomly-generated $100{,}000$-element test set.

The last two rows of Table 2 show that input feature subsets F and H lead to the fastest average search times for LightGBM and linear regression, respectively. Subset F uses all graph features except the most expensive, island count. Its effectiveness demonstrates the value of the compatibility graph, despite its cost. Subset H also performs well in linear regression, perhaps suggesting that the three features in subset H are most important of all, and that the others in subset F require a more sophisticated model to maximize utility.

Algorithm Behavior in Ordered DFS Given the results of the previous experiment, we explore further LightGBM with feature subset F and linear regression with feature subset H. Additional data are provided in Table 3. Before considering these results, several facts must be established.

			A	B	C	D	E	F	G	H	I	J	K	L
Card	RC count	min, max	✓	✓	✓									
		median, avg	✓	✓										
	RC compat	min, max	✓	✓	✓							✓	✓	
		median, avg	✓	✓								✓	✓	
	count	ace, king	✓	✓	✓							✓	✓	
		suit ratios	✓	✓	✓							✓	✓	
		rank stdev	✓											
Graph		islands	✓			✓	✓					✓		✓
		critical	✓			✓	✓	✓	✓			✓		✓
	degree	min	✓			✓	✓	✓	✓	✓		✓		✓
		max	✓			✓	✓	✓	✓	✓				
		median, avg	✓			✓		✓						
	conn.	node conn.	✓			✓	✓	✓	✓	✓	✓	✓		✓
		is conn.	✓			✓	✓	✓	✓			✓		✓
Total time (seconds)		Linear Reg	0.672	0.333	0.322	0.347	0.410	0.281	0.335	**0.254**	0.324	0.542	0.371	0.516
		LightGBM	0.690	0.442	0.416	0.412	0.491	**0.333**	0.408	0.345	0.461	0.724	0.544	0.679

Table 2: 12 subsets of features and average total search time for each subset

	Lin. Reg., H	LightGBM, F
Pred. Time ($\times 10^{-5}$ s)	0.016	0.578
RMSE	0.400	0.256
Node Count	1381	1045
Total Time (s)	0.254	0.333

Table 3: Additional performance data for linear regression (subset H) and LightGBM (subset F).

The prediction time and RMSE rows of Table 3 correspond to results about the estimation models themselves, separate from Ordered DFS. Each model was tested via 10-fold cross validation on the same $500{,}000$-element randomly-generated dataset. Prediction time refers to the average time to estimate solvability per state. RMSE refers to the average root-mean-squared error in estimating solvability, where the estimated value is compared to a target value of 1 for solvable states and 0 for unsolvable states. Thus, for RMSE a lower number is better. Note that a prediction of "unsolvable" for every state results in an RMSE of 0.698 for this dataset. Also note that this RMSE measure for regressors is distinct from the accuracy measure for classifiers used in the Solvability Model discussion above.

The node count and total time rows of Table 3 correspond to results about the application of each of the estimation models to Ordered DFS. After training each estimation model on the same $500{,}000$-element random dataset, the resulting Ordered DFS system conducts a search on each element in a separate randomly-generated $100{,}000$-element test set. The node count corresponds to the average number of nodes that are "processed" – that is, the number of nodes checked for being terminal or having a disconnected compatibility graph, followed by node expansion if warranted. A lower node count means more effective pruning has occured. The total time column is the average total amount of time that an Ordered DFS search requires.

It is also important to note that prediction time $\times$ node count $\neq$ total time, for two reasons. First, there is additional overhead beyond just prediction time that is not captured in Table 3. More importantly, though, note that predictions are made for far more nodes than are actually counted in the "node count" measure. Predictions must be made for every generated child to create an ordering. If the ordering is effective and a solvable child is found quickly, then most of those children will be pruned (thus not "counted"), while a prediction was nevertheless determined for each.

Having established the meanings of these measures, consider the results in Table 3. LightGBM has a lower average RMSE than linear regression, at 0.256. This is not surprising, given the increased sophistication of LightGBM over linear regression. Lower error means more effective ordering, and thus more pruning. This is reflected in the fact that LightGBM also has a lower average node count than linear regression, at 1045. However, it also is not surprising that this low error requires a higher prediction time per node, at 0.578×10^{-5} s.

Linear regression has a worse RMSE and therefore a higher node count than LightGBM. Its prediction time, on the other hand, is much faster. This faster prediction time results in a faster total time, despite the higher node count. Since linear regression on feature subset H is slightly faster than LightGBM on subset F, we consider further the linear regression model in Ordered DFS in the next experiment.

Ordering Cutoff Tuning Despite the results above, even a linear regression model on subset H applied to Ordered DFS is slower than Compatible DFS, with average total times of 0.252s and 0.140s, respectively. In an effort to improve Ordered DFS's results, this next experiment considers the selective application of ordering. Note that ordering of siblings is less useful in late-game states than early-game states. Late-game states have trees with fewer levels, and so the benefits of pruning siblings decreases, while the solvability estimation cost per child remains the same. A new *ordering cutoff* hyperparameter allows more selective application of sibling ordering, by specifying the number of stacks in a state for which such ordering will no longer be done. For example, with an ordering cutoff value of 5, Ordered DFS

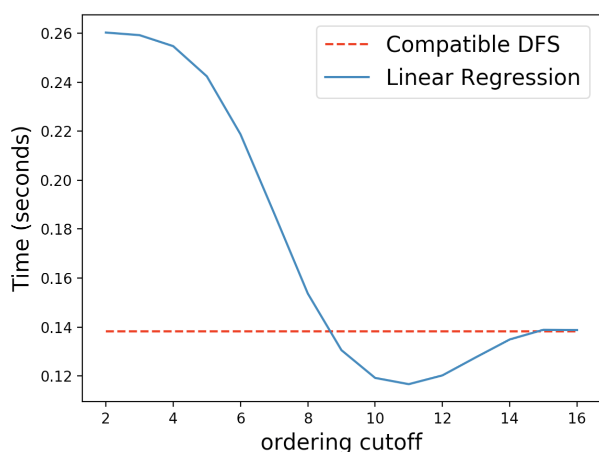

Figure 2: Effect of ordering cutoff on Ordered DFS average total time

performs sibling ordering only for states with greater than five stacks. Otherwise, an arbitrary ordering is used, and no solvability estimation cost is incurred – just as in Compatible DFS.

The lowest sensible ordering cutoff value is 2, meaning that all children will be ordered except for children of two-stack states. It never makes sense to order the children of a two-stack state, as would be done with ordering cutoff 1. This is because all two-stack states have either zero or two children. Zero children means the two-stack state is an unsolvable state, and of course there are no children to order. Two children means the two-stack state is solvable by putting either stack on the other, with the only difference being which stack ends up on top. Thus, ordering of these two states is irrelevant.

In this experiment, then, 15 Ordered DFS systems are considered, with ordering cutoffs from 2 to 16. Other characteristics of the systems use the conclusions from previous experiments. Each uses linear regression, since that was the best model in the previous experiment. Each also uses feature subset H – the best subset found for linear regression in the first Ordered DFS experiment.

Results are in Figure 2. The curve shows the average search time for Ordered DFS with linear regression for each ordering cutoff. For comparison, the straight line shows the performance of Compatible DFS, which does not consider ordering cutoff at all. Note, as expected, that Ordered DFS with ordering cutoff 16 is equivalent to Compatible DFS, plus a bit of overhead. With ordering cutoffs below 9, the tree is small enough that the effort spent ordering does not lead to significant enough pruning to be worthwhile. The ideal ordering cutoff is 11.

Based on this series of experiments, the ideal Ordered DFS system uses linear regression with feature subset H and ordering cutoff 11. This system is used in the comparison experiments described below.

Island Merge

Another algorithm that uses graph features for pruning is Island Merge. Recall the definitions of exterior and interior island, critical node, and critical edge, as illustrated in Figure 1. Islands are important because with the exception of critical nodes, to maintain solvability requires that stacks in an island be moved to cover only other stacks in the same island. Critical nodes and edges must be handled correctly to avoid a disconnected compatibility graph and therefore unsolvable state.

Island Merge is a search based on Compatible DFS, but with further pruning based on compatibility graph analysis. As gameplay progresses, more stacks are covered, and thus more nodes are removed from the compatibility graph. This means fewer edges in the compatibility graph and a greater chance of critical nodes and edges. Island Merge tracks these graph features. While we do not provide here a formal proof of correctness, the algorithm described below achieves 100% accuracy on a test set of 500,000 randomly-generated states.

The key idea of Island Merge is that each island can be treated as a "mini-game" of BoaF. To *collapse* an island is to make a series of moves reducing that island to a single stack, essentially solving that mini-game. In the same way that a full BoaF game state may be unsolvable even with a connected compatibility graph, an island may also be uncollapsible even with a connected compatibility (sub)graph. Thus, an attempted collapse operation may fail.

Exterior islands should be collapsed before interior islands. This is because an interior island typically has at least two critical nodes; the process of collapsing will inherently cover one of them, thereby severing the connection to the corresponding island. Thus, Island Merge prunes the tree such that only exterior island-collapsing moves are considered. In rare circumstances, an interior island could be collapsed first if it has just one critical node with multiple critical edges, but exterior islands may always be collapsed first.

In addition to collapsing, Island Merge uses a second operation, *merging*. Suppose s is a single-stack exterior island adjacent to another island a. To merge s into a is to consider s as part of a when collapsing a. If a is an interior island connected to a set of islands E, then a becomes an exterior island when all but one island in E is merged with a. In short, under the stated conditions, merging converts an island from interior to exterior, making it eligible for collapsing.

At a high level, then, Island Merge chooses between these two operations: to collapse an exterior island, or to merge a single-stack exterior island with another island. If a collapse operation is chosen, then moves not corresponding to collapsing that island can be pruned, until the collapse is complete or it fails. If a merge operation is chosen, then the single-stack exterior island is considered part of another island, and then a new high-level operation is chosen.

This exploration of high-level operations can be modeled as a search in an *island tree*, as illustrated in Figure 3. Initially, the game tree is explored via ordinary Compatible DFS, as shown with the "DFS" marker at the top of the tree. This continues with each new state generated by a player move, until the current game state comes to have

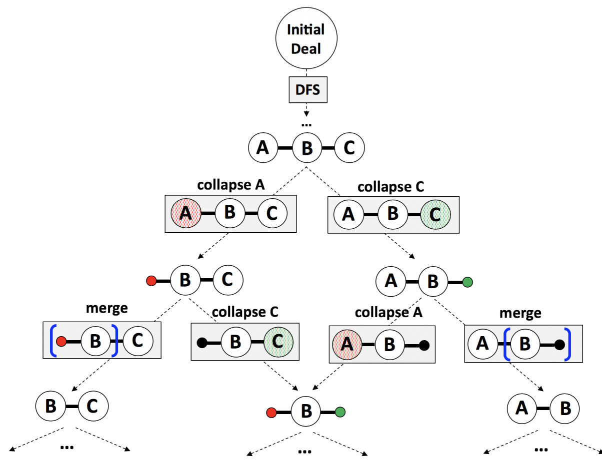

Figure 3: Game tree for island merge

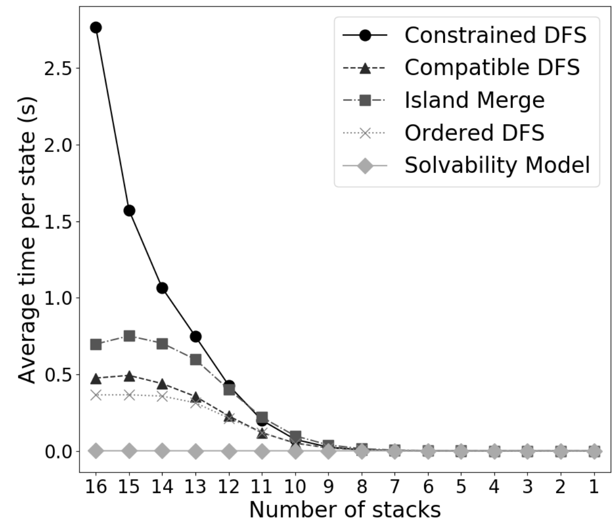

Figure 4: Comparison of the average runtime for each algorithm.

multiple islands. At this time, Island Merge chooses from available high-level operations. For example, in Figure 3, we show three islands near the root of the tree, indicated by Ⓐ – Ⓑ – Ⓒ. Note that B is an interior island, while A and C are exterior islands, and so the algorithm can choose to collapse either A or C. Suppose the algorithm chooses to attempt to collapse A. In that case, a successful collapse (if it exists) is found and the algorithm moves on to the next high-level decision. At that point, it either merges A into B or collapses C. If required due to exclusively unsolvable states further down the tree, the algorithm backtracks to find an alternative successful collapse of A (if it exists). If no successful collapses of A exist, or none lead to ultimately solving the game, then backtracking occurs one level higher in the island tree, such that the algorithm attempts to collapse C instead of A. This process continues until either a solved state is found and the search returns "solvable", or all moves result in unsolvable states, and the search returns "unsolvable". Below, the performance of Island Merge is compared with the other algorithms of this paper.

Algorithm Comparison

Each algorithm is evaluated based on runtime and nodes searched, using a dataset of 248,170 randomly-generated states. All tests are conducted using a local parallel computing network of 20 computers running Ubuntu 16.04 with a 4-core Intel i5-2400 CPU and 8GB of RAM each.

Figures 4 and 5 illustrate the mean performance of each algorithm in terms of runtime and number of nodes searched, respectively. These data suggest that the number of nodes explored is the greatest influence on runtime. They also demonstrate improvements by all experimental algorithms, both in runtime and nodes searched, relative to Constrained DFS. This advantage tapers off as the number of stacks in the state decreases, to the point where differences in performance between algorithms are comparatively small. Thus, the data can be separated into two major regions: the divergent region concerning states with at least ten stacks, and the convergent region concerning states with less than ten stacks.

Consider Figure 4. Solvability Model (with LightGBM) has the fastest running time in the divergent region, at 1.9822×10^{-6} seconds. Note, however, that Solvability Model has imperfect accuracy, at 94.479%. Among the 100% accuracy algorithms, Ordered DFS consistently has the fastest runtime in the divergent region. Compatible DFS has the second-fastest runtime, followed by Island Merge. In the convergent region, however, the time needed to search the game tree is less than that required to apply Solvability Model's prediction strategy. Thus, the Solvability Model is the slowest algorithm in that region, but only by a very small margin. Furthermore, since Constrained DFS has the fastest per-node runtime, it has the fastest overall runtime in the convergent region; again, though, the difference is very small.

Figure 5 highlights the fact that Island Merge searches the fewest nodes in the divergent region. Ordered DFS comes in a close second, followed by Compatible DFS. The Solvability Model cannot be considered in this comparison, since it does not explore the game tree. Even though Island Merge has the highest per-node runtime, its reduced number of nodes explored suggests that it may have the fastest potential runtime of all experimental algorithms. Additionally, all three experimental algorithms search nine to ten times fewer nodes than Constrained DFS on average.

In Figure 5, it is interesting to note the small rise in nodes examined for Constrained DFS for lower numbers of stacks. This is because the fewer stacks a randomly-generated state has, the less likely that it is solvable. Since Constrained DFS prunes only when a solvable node is found, unsolvable states will result in no pruning, and thus a higher count of nodes examined. This does not occur in the other algorithms, however, because their pruning is based on not only solvability, but also the compatibility graph.

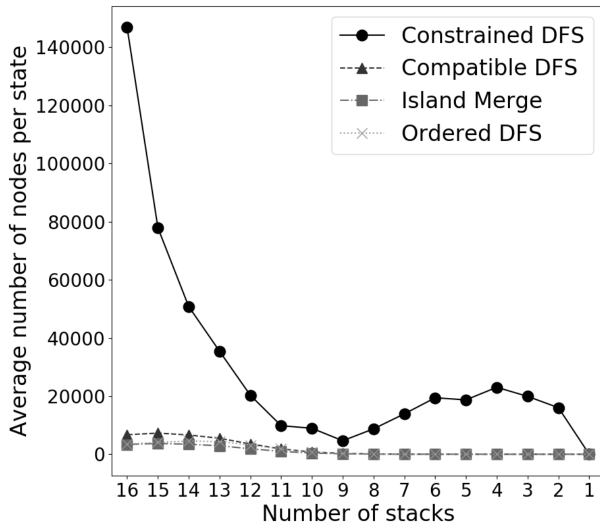

Figure 5: Comparison of the average number of nodes explored for each algorithm.

Considering both figures, there is a comparatively large difference between the three experimental algorithms and Constrained DFS – but is this difference significant? Since the data used to generate Figures 4 and 5 is not normal, we cannot use standard deviation to determine statistical significance. However we use a T-test to show that each algorithm is significantly different from the others, since the sample size is statistically large. Specifically, the largest p-value among all pairwise combinations of algorithms in the figures is less than 0.0001, suggesting statistical significance.

Interestingly, while Compatible DFS is not the fastest algorithm, its overall performance is comparable to Ordered DFS and Island Merge. The reasonably strong performance of Compatible DFS, which checks only for compatibility graph connectivity, suggests that this feature is critical in developing an efficient search-based classifier.

Related Work

Games frequently serve as the context for developing optimization algorithms, since many are NP-complete despite having a simple rule set. That is, their large search space makes an exhaustive search of potential moves intractable. This prompts the development of heuristics and pruning strategies to maximize efficiency with minimal losses to accuracy. BoaF is an example of a game with *perfect information*, since no relevant game information is hidden from the player. Since BoaF is newly-developed, no research exists particularly for this game, but much related work exists for other perfect information games.

Solver algorithms are designed to maximize play accuracy for a multi-dimensional problem, since a successful game is often determined by more than one characteristic. This idea of a nonlinear solution-finding process highlights the need for *planning* on the part of the solver (Hoffman and Nebel 2001). Multidimensional planning is most evident in multi-player games such as checkers (Samuel 1959), chess (Campbell, Hoane Jr., and Hsu 2002), and Go (Silver et al. 2016), since the strategy of the solver is dependent upon an unknown sequence of opposing moves. However, planning is still necessary in relaxed, single-player versions of these games, as well as games such as FreeCell (Paul and Helmert 2016; Elyasaf, Hauptman, and Sipper 2011) and solitaire (Bjarnason, Tadepalli, and Fern 2007; Helmstetter and Cazenave 2003), where optimal play may be defined by various independent factors. BoaF may also be included in this group, as many similarly independent properties of stacks exist. These properties influence branching in the game tree and in the path to a solution.

However, BoaF differs significantly in that the number of moves for each solution is identical. That is, while cycling through the deck in solitaire and using free cells in FreeCell lead to solutions of varying length, every solution to a solvable BoaF state with n stacks is $n - 1$ moves long. As a result, there is less emphasis in BoaF on efficient moves as there is on accurate moves, and algorithms accordingly focus more on intelligent move selection and pruning of the search space. This logic relates to work on deadlock patterns by Paul and Helmert (2016), and is strikingly similar to ideas proposed by Helmstetter and Cazenave (2003). In fact, many parallels exist in limitations and patterns based on state positions in BoaF and Gaps (Helmstetter and Cazenave 2003).

Conclusions and Future Work

This research presents several BoaF solvability classifier algorithms. The most effective algorithm tested is Ordered DFS with linear regression, feature subset H (a few key graph features), and ordering cutoff 11. Other tested strategies are nearly as effective, and all show significant improvement over the baseline Constrained DFS approach in runtime and search space size.

These advancements also open various avenues for further research. Potential further optimizations exist for many algorithms, such as repeated island detection in Island Merge. Additionally, features that would require an exhaustive search to generate may be estimated via efficient exploration of the search space. For example, score is calculated as the square of the number of cards in each stack in a terminal or unsolvable state. Thus, mean score for a state may be estimated via an Island Merge-based approach, as island size could predict maximum stack sizes for unsolvable states. Score metrics could inform evaluation of state difficulty, and/or evolutionary generation of difficult states. Testing states with different row/column dimensions could encourage novel algorithm strategies, owing to different search space sizes, compatibility graph connectivity, and/or positional limitations. Lastly, a greedy solver algorithm is a natural extension of the proposed search algorithms, as they find the solution of any state declared solvable. Focusing on a solver could employ score-based decisions, as well as elements of Monte-Carlo tree search, to build on intelligent ordering and pruning methods outlined in this paper.

References

Bjarnason, R.; Tadepalli, P.; and Fern, A. 2007. Searching solitaire in real time. *ICGA* 131–142.

Campbell, M.; Hoane Jr., A. J.; and Hsu, F.-h. 2002. Deep blue. *Artificial Intelligence* 134(2):57–83.

Elyasaf, A.; Hauptman, A.; and Sipper, M. 2011. GA-FreeCell: evolving solvers for the game of FreeCell. In *13th Annual Genetic and Evolutionary Computation Conference*, GECCO '11, 1931–1938.

Helmstetter, B., and Cazenave, T. 2003. Searching with analysis of dependencies in a solitaire card game.

Hoffman, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Ke, G.; Meng, Q.; Finley, T.; Wang, T.; Chen, W.; Ma, W.; Ye, Q.; and Liu, T.-Y. 2017. Lightgbm: A highly efficient gradient boosting decision tree. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc. 3146–3154.

Kohavi, R. 1995. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'95, 1137–1143. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Li, P.; Wu, Q.; and Burges, C. J. 2008. Mcrank: Learning to rank using multiple classification and gradient boosting. In *Advances in neural information processing systems*, 897–904.

Neller, T. 2018. "birds of a feather" solitaire card game. http://cs.gettysburg.edu/ tneller/puzzles/boaf/. Accessed: 2018-07-11.

Paul, G., and Helmert, M. 2016. Optimal solitaire game solutions using A* search and deadlock analysis. In *Proceedings of the Ninth International Symposium on Combinatorial Search*, SoCS 2016, 135–136.

Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; Vanderplas, J.; Passos, A.; Cournapeau, D.; Brucher, M.; Perrot, M.; and Duchesnay, E. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12:2825–2830.

Ranka, S., and Singh, V. 1998. Clouds: A decision tree classifier for large datasets. In *Proceedings of the 4th Knowledge Discovery and Data Mining Conference*, 2–8.

Samuel, A. L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of research and development* 3(3):210–229.

Shi, H. 2007. *Best-first decision tree learning*. Ph.D. Dissertation, The University of Waikato.

Silver, D.; Huang, A.; Maddison, C.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. 529:484–489.