Efficient Solving of Birds of a Feather Puzzles

Todd W. Neller and **Connor Berson** and **Jivan Kharel** and **Ryan Smolik**
Gettysburg College
{tneller,bersco01,kharji01,smolry01}@gettysburg.edu

Abstract

In this article, we describe the lessons learned in creating an efficient solver for the solitaire game Birds of a Feather. We introduce a new variant of depth-first search that we call *best-n depth-first search* that achieved a 99.56% reduction in search time over 100,000 puzzle seeds. We evaluate a number of potential node-ordering search features and pruning tests, perform an analysis of solvability prediction with such search features, and consider possible future research directions suggested by the most computationally expensive puzzle seeds encountered in our testing.

Introduction

Birds of a Feather (Neller 2016) is an original perfect-information solitaire game played with a standard 52-card deck. After shuffling, the player deals the cards face-up into an r-by-c grid of cards. In this paper we focus on the case where $r = c = 4$.

The object of Birds of a Feather is to move all grid cards into a single stack. Think of each grid cell as initially containing a 1-card stack. A stack may be moved on top of another stack in the same row or same column if the top cards of the two stacks have the same suit, the same rank, or adjacent ranks. Aces and kings are considered low and high, respectively, and are non-adjacent to each other.

For example, consider the initial game state shown in Figure 1(a). Having the same rank in the same column, the JS (Jack of Spades) stack can move onto the JC (Jack of Clubs) (Fig. 1(b)). Having adjacent rank in the same column, the TS (Ten of Spades) stack can move onto the 9H (9 of Hearts) (Fig. 1(c)). Having the same suit in the same row, the JS stack can move onto the 5S (Fig. 1(c)). We can solve this deal, moving all cards into a single stack, with this sequence of moves: JS→JC, TS→9H, JS→5S, KS→3S, KS→KC, JS→KS, JS→TS, 6H→7D, 6H→5C, 6H→8H, QH→AH, QH→TH, QH→3H, QH→JS, QH→6H.

In this work, our objective was to create an efficient search algorithm that would enable us to speedily find a puzzle solution or, through unsuccessful complete search, demonstrate that no solution exists.

We begin by defining search feature terms used in our experimental work. Next, we evaluate the search time performance of various search algorithms. A state solvability dataset is then described and used to evaluate our defined search features, both singly and in combination with one another as suggested by regression of solvability onto feature subsets. Next, we turn attention to the use of features for accurate prediction of state solvability. Finally, we examine some of the puzzle seeds that were most computationally expensive for our best solver and consider future research directions suggested by observations of such puzzles.

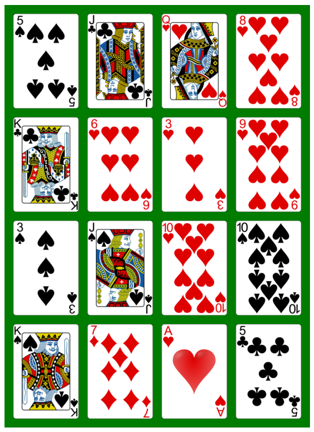
Search Features

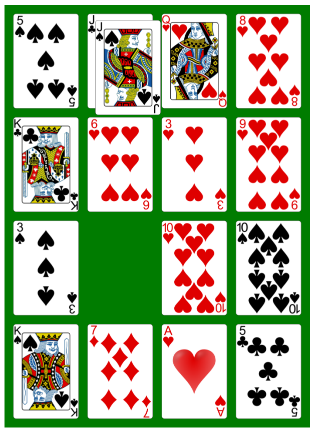
Our baseline for search comparison is the performance of iterative, stack-based, depth-first search (DFS). Before describing our variations and experimental results, we first need to define the terms and features we make use of in search pruning and node ordering.

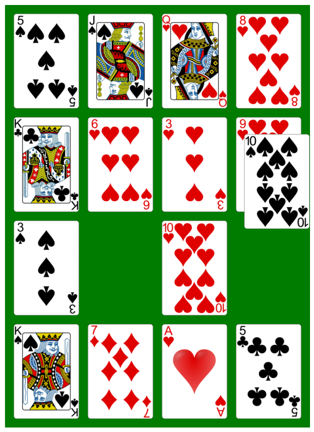
Two cards are said to be *flockable* if they have the same suit, the same rank, or adjacent ranks. A *flockability graph* is a graph where each node corresponds to the top card of a stack, and there is an edge between two nodes if and only if the two corresponding cards are flockable. We will refer to the total number of flockability graph edges as a measure of *total flockability* of the game state, which we will also refer to as *state flockability*. *Card flockability* is the degree of a card's node in the flockability graph. Flockability is different from number of legal moves since we ignore same-row/-column constraints in flockability in order to gain an expectation of the number of legal moves in future states, i.e. the Chess game concept of *mobility*.

The *flockability graph* structure can indicate unsolvability of a state. We define an *odd bird* to be a card on the game grid that can never be flocked with another card, i.e. having a flockability graph node degree of 0. A flockability graph with an odd bird indicates an unsolvable game state. More generally, a *separated flock* exists when the flockability graph has more than one graph component, i.e. two or more disjoint sets of cards can never flock with each other.

Another flockability graph feature of interest was its *minimum degree*, i.e. the number of cards flockable with the least flockable card. In the extreme case with a minimum degree of 0, we have an unsolvable *odd bird* state. Thus, one would expect states with a greater minimum degree to be


(a) Initial state


(b) After JS→JC


(c) After TS→9H

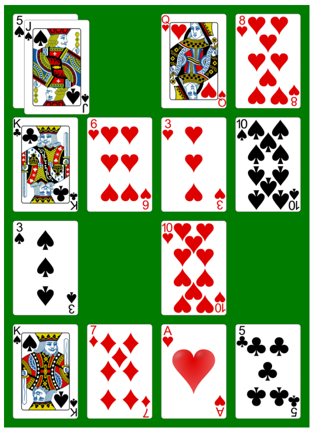

(d) After JS→5S

Figure 1: Example Birds of a Feather moves

more likely to be solvable, presenting more future options for moving any given card.

Another measure more specific than flockability is the number of legal move card pairs each state has, referred to as state *mobility*. One can also think of mobility as the number of edges on the mobility subgraph of the flockability graph. The *mobility per card* feature provides a measure of relative mobility that scales across play from opening to endgame. We also make use of what we call the *mobility-flockability ratio* of a state, defined as the ratio of *mobility* to *flockability*. In other words, a state's flockability ratio is the fraction of flockable pairs that share the same row or column.

The feature *rank cluster count* is the number of groups of contiguous adjacent ranks that exist among the remaining cards. If one plots a histogram of occurring ranks, the rank cluster count is the number of disjoint contiguous ranges of occurring ranks. This feature provides a rough measure of how much we must rely on shared suits to flock remaining cards. For example, the board in Figure 1(a) has a *rank cluster count* of 3. This initial state has all ranks except 2 and 4. This separates the Ace from the 3 and the 3 from the large, contiguous rank cluster of 5 through King.

The feature *rank deviation* is the standard deviation of the rank distribution when representing successive ranks as successive integers. Whereas rank cluster count gives us a measure of contiguity for flocking among ranks, rank deviation gives us an idea of how we are progressing at eliminating extreme ranks and working towards a single narrow range of ranks.

The *suit dominance* feature is the difference of fractions of cards occurring in the most- and least-frequently occurring suits. For example, if the most- and least-frequently occurring suits are clubs and diamonds with 4 cards and 1 card, respectively, out of a total of 10 cards, then the suit dominance feature would be $\frac{4}{10} - \frac{1}{10} = \frac{3}{10}$. The least-frequently occurring suit fraction need not be non-zero; if a suit is not represented, i.e. has a fraction of 0, we still treat that as the least-frequently occurring suit. This helps to determine the dominance of one suit over others on the board.

One of the more computationally complex features computed was the state *flockability diameter*, hereafter referred to as the *diameter*. *Diameter* is the maximum eccentricity of any vertex in the flockability graph, which can be thought of as the largest minimum number of moves needed to flock two cards together. The diameter of the flockability graph was computed as the maximum entry in the all-pairs shortest-paths distance matrix of the Floyd-Warshall algorithm (Cormen et al. 2009, §25.2).

Comparing Search Algorithms

As a search problem, Birds of a Feather puzzles have a few important characteristics. First, repeated state detection is important. Combinatorially many paths may lead to the same state, so efficient search will cache states known to be unsolvable. Second, whereas naïve uninformed search would run into memory problems with such caching, judicious pruning allows us to prove states unsolvable without excessive memory requirements. Third, all goal nodes of search, i.e. single-stack solutions, are found at the same depth, as with classic Peg Solitaire puzzles (Beasley 1985). Thus, depth-/cost-minimizing search algorithms such as A*, recursive best first search (Korf 1993), iterative deepening, or similar search algorithms (Russell and Norvig 2009) are not relevant for this problem domain. Of greater interest for this domain is the prioritization of search nodes that are more likely to be solvable.

Given such search problem characteristics, we implemented an iterative, stack-based depth-first search, a recursive, lazy-evaluation depth-first search, a best-first search, a depth-first search with node ordering, and a search algorithm that we refer to as *best-n depth-first search*.

Our search algorithm pruning came through combinations of three different methods: (1) caching searched states and avoiding repeated states, (2) eliminating odd bird states, and (3) eliminating separated flock states. All of our search algorithms make use of (1), hashing states that are unsolvable. We will see the impact of the latter two pruning methods with different search algorithms.

We check for odd birds by initializing three lists prior to search, each indexed by an integer card identification number. The *flockabilityLists* is a list of lists of card numbers. The list at a card ID index contains all other cards flockable with that card. This is immutable throughout search. A boolean array *isPresentArray* is true at a card ID index if and only if that card is currently on the board. The most important structure for our pruning computations is an array of integers named *flockability*. This array is initialized as each card's flockability. All cards that are not currently on the board are initialized with value -1. By using the *flockabilityLists* and the *isPresentArray*, we can easily update the *flockability* array throughout search. If a card is no longer present on the board, its value becomes -1 and all cards flockable with that card has its *flockability* array decremented. A zero value in the *flockability* array thus indicates the presence of an odd bird.

Basic Depth-First Search: Using the odd bird feature, children of nodes were pruned upon expansion into the next generation. On a test of 15,000 seeds, the inclusion of our odd bird check decreased the basic DFS code's average solve time per seed from 1,212 ms to 133 ms. Even with the utilization of the odd bird check on root nodes only, the average solve time per seed decreases from 1,212 ms to 556 ms.

Since separated flock states are unsolvable, we can prune such states in search. Our average search time per seed over 15,000 seeds decreased from 1,212 ms to 61.5 ms when children were pruned based on separated flocks only. Similarly, when only root nodes were pruned for separated flocks, the average time was 502 ms per seed. Specific seeds proved difficult for our algorithm to solve prior to introducing the separated flocks check. For example, an instance of separated flocks, seed 1,264, took 192,185 ms to complete the unsolvable search. After introducing the separated flock check and hybrid search, the time taken for this unsolvable search was negligible (< 1 ms).

The individual use of the odd bird check and separated flocks checks each provided large improvements to the performance of our basic DFS search algorithm. However, when these checks were utilized together, another decrease was found, albeit a smaller improvement. Separated flock checking subsumes simpler odd bird checking, so one might think that performing both would be redundant and detrimental to search, yet combination of both checks with odd bird checking first and short-circuit evaluation decreased the average time per seed slightly to 59.8 ms.

Recursive Depth-first Search: One improvement we can make on iterative stack-based depth-first search is to perform lazy evaluation on node expansion and implement it as a recursive depth-first search that searches non-pruned children as they are generated, potentially saving the computational cost of fully expanding a node, generating children that are never searched. Whereas basic DFS averaged 1,212 ms per seed, this improvement reduced average search time to 143 ms over 100,000 seeds.

Best-First Search: Our work so far has focused on variants of depth-first search algorithms that have improved our search time. A different use of guided search methods was considered in our work by applying a best-first search algorithm to our problem. By both pruning our nodes for unsolvable states and ordering our nodes based on the ratio of *flockability* to number of cards, we used this feature to guide the best-first search, leading it to an average search time of 141 ms over 100,000 seeds. The use of best-first search resulted in a slower speed when compared to improvements to the basic depth-first search algorithm. Observing better performance of basic depth-first search with pruning, we subsequently limited our focus to depth-first search algorithms.

Depth-first Search with Node Ordering: We next experimented with a node ordering heuristic with pruning. This algorithm, referred to here as *DFS with node ordering*, sorted children based on *state flockability*. A base test for DFS with node ordering and pruning based only on a set of cached states was too slow to allow for adequate data collection. When odd bird pruning was added, the average time dropped to 50.17 ms. The use of the separated flocks check individually also lowered the average time to 13.58 ms. The use of both features combined with short-circuit evaluation lowered the average time to 10.06 ms.

Best-n Depth-first Search: We next created a variant of depth-first search called *best-n depth-first search* that differs from a straightforward depth-first search with heuristic node-ordering by allowing alternative backtracking order that may include local unsearched ancestors. This algorithm is described as Algorithm 1.

Best-n depth-first search maintains a global search stack S to manage backtracking and ensure search completeness as usual, but manages forward search in small batches we will term *cohorts* that are queued in Q. When a non-closed, non-goal node is removed from Q, it is expanded and its unpruned children are sorted via priority queue PQ according to a node-ordering metric and placed on S. Cohorts are formed by popping up to n nodes from S that have the same heuristic value as the top stack node. These are enqueued in Q and search continues until Q and S are empty. In our implementation, we set $n = 4$.

Note that a cohort may include nodes of different depths that share a common heuristic value. Since these nodes as a group have their unpruned children added to the global stack, and the last node from the queue may be of a lesser depth than the first, search order is not strictly depth-first, but rather depth-first by cohort. A cohort's children are completely searched before backtracking via the search stack.

As before, we prune nodes that have been already searched, are odd birds, or are separated flocks. Pruning techniques when used in the best-n DFS algorithm resulted in further search time reductions from DFS with node ordering. When the odd bird test is utilized on 15,000 seeds, the average time decreased to 42.41 ms. Furthermore, the use of the separated flocks as a pruning technique decreased the average time to 7.01 ms. The use of both odd bird and separated flocks for pruning resulted our lowest average time at 5.32 ms. From the basic DFS average time per seed of 1,212 ms, best-n DFS with both odd bird and separated flock pruning reduced average time per seed to 5.32 ms, a 99.56%

Algorithm 1 Best-n Depth-First Search

```
 1: Input: root search node, cohort size $n$
 2: Output: whether or not a goal node is found
 3:
 4: function search($rootNode$, $n$):
 5: $Closed \leftarrow \{\}$ {set of searched states}
 6: $Q \leftarrow [rootNode]$ {cohort search queue}
 7: $S \leftarrow []$ {global search stack}
 8: $PQ \leftarrow []$ {cohort children priority queue}
 9: while $Q$ not empty do
10:     while $Q$ not empty do
11:         $node \leftarrow Q$.dequeue()
12:         if $node \notin Closed$ then
13:             $Closed \leftarrow Closed \cup \{node\}$
14:             if $node$ is a goal node then
15:                 return true
16:             end if
17:             enqueue into $PQ$ unpruned children of $node$
18:         end if
19:     end while
20:     while $PQ$ not empty do
21:         $S$.push($PQ$.dequeue()) {leaving heuristically pre-
            ferred children of $Q$ cohort on top of $S$}
22:     end while
23:     if $S$ not empty then
24:         $v \leftarrow$ heuristic value of $S$ top node
25:         enqueue into $Q$ cohort of up to $n$ popped items of
            $S$ with a heuristic value equal to $v$
26:     end if
27: end while
28: return false
```

Search Algorithm	$\frac{\textbf{Time (ms)}}{\textbf{seed}}$
Basic DFS	1212
... with odd bird pruning	133
... with separated flock pruning	61.5
... with both prunings	59.8
Recursive DFS	143
Best-First Search (Flockability/Card)	141
DFS with node ordering, odd bird	50.17
... with separated flock pruning	13.58
... with both prunings	10.06
Best-n DFS with odd bird pruning	42.41
... with separated flock pruning	7.01
Best-n DFS with both prunings	5.32

Table 1: Search algorithm time per seed for 100,000 seeds, except for the following algorithms tested with 15,000 seeds: basic DFS, DFS with node ordering and odd bird pruning, and best-n DFS with odd bird pruning.

reduction.

Solvability Dataset

In future experimental sections, we made use of a dataset of 1,230,606 Birds of a Feather non-terminal game states labeled with 0 (unsolvable) or 1 (solvable). These were generated from 10,000 puzzle seeds starting with seed 1100. For a given seed, the initial state is searched, labeled as solvable or not, and added to the dataset. If the puzzle is not solvable, we then proceed to the next puzzle seed.

For solvable puzzle seeds, we perform the following iterative process. For each state, we generate all children, search all children and label each as solvable or not. If there exists an unsolvable state among the children, all children and labels are added to the dataset. However, if all children are solvable, the decision at that state is not important, so we exclude them from the dataset. A random solvable child is then chosen as the next state. This process is repeated until the solvable puzzle is solved.

Through such selective random sampling, we achieve a reasonable representative sample of solvable and unsolvable game states that one might be presented with in critical decisions while playing through 10,000 puzzles. We found that this sampling method also achieved a reasonably balanced sample of 572,718 solvable states and 657,888 unsolvable states.

Feature Selection for Solvability Prediction

Overall Feature Performance

One of our research foci was to determine the performance of subsets of features for the prediction of state solvability using logistic regression. We were also interested to determine which single feature best predicts solvability of Birds of a Feather game states. In order to determine best feature subsets, we took the aforementioned solvability dataset and augmented each entry with features normalized by $[0, 1]$-scaling. We then utilized exhaustive selection of feature subsets, performing logistic regression on each subset to predict solvability. Among these, we observed which feature subset and which single feature minimized p-values, suggesting a strong prediction of solvability.

Our best subset of features for solvability prediction is the combination of *mobility per card*, *state flockability*, *mobility*, *mobility-flockability ratio*, *minimum degree of flockability*, *rank cluster count*, *rank deviation*, *suit dominance*, and *diameter*. Logistic regression on this specific feature subset followed by rounding to a 1 (solvable) or 0 (unsolvable) prediction gives us an accuracy of 75.45%. This was a 1% improvement over what we had with these same features but excluding *diameter*.

Although we were surprised to see that *diameter* alone did not prove to be a significant predictor of solvability, it yielded some consistency in exhaustive feature search. Prior to introducing *diameter* the predictions resulting from stepwise selection on sorted depths varied between *mobility per card*, *minimum degree*, and *flockability*. After introducing diameter as a feature, this variation no longer existed and

Features	Coef.	z-value	pr(>$\|z\|$)
Intercept	-4.498	-234.552	< 2e-16
Mobility per card	7.678	197.347	< 2e-16
State flockability	0.062	141.963	< 2e-16
Mobility	-5.659	-68.492	< 2e-16
Mobility-flockability ratio	3.104	117.893	< 2e-16
Minimum degree	1.133	44.359	< 2e-16
Rank cluster count	-1.829	-112.133	< 2e-16
Rank deviation	-0.449	-21.253	< 2e-16
Suit dominance	2.875	162.962	< 2e-16
Flockability diameter	-0.019	-0.766	0.444

Table 2: Coefficients of features used for logistic regression yielding our best accuracy of 75.45%.

mobility per card was predicted as the best single feature at each depth.

Surprisingly, our best single feature for prediction of solvability was *mobility per card*. This contradicted our original hypothesis that *state flockability* would be the best feature for predicting solvability. Solvability prediction using *mobility per card* alone as a feature resulted in an accuracy of 70.64%. Comparing this to our best feature combination accuracy of 75.45% above, we see that *mobility per card* accounts for much of the accuracy attainable with these features.

Feature Performance by Depth

We became curious if *mobility per card* still offered the best feature for prediction of solvability when regressing separately for each node depth. In order to obtain such information, we partitioned our dataset by number of cards, i.e. with different depths from 2 to 14. We then used exhaustive search for feature selection at each depth, giving us information on best features at each depth. We found that *mobility per card* was still the best feature for all depths.

Before we introduced *diameter*, *state flockability* was most frequently the best single feature that appeared on stepwise selection. It is also important to note that for a majority of depths it was *state flockability* that turned out to be the second best single feature as suggested by exhaustive search. Although *state flockability* was not found to be the best single feature as frequently as it had been without the inclusion of the *diameter* feature, it remained one of the our prominent predictors correlating with solvability.

We were also interested in finding whether there existed features that would act in contrast at differing depths when we regressed solvability onto them. Our answer to the question lay in the effect that the *suit dominance* had on solvability at different depths. For prediction, we rounded the output of logistic regression to 1 (solvable) and 0 (unsolvable). We found that the feature *suit dominance* had a positive coefficient on regressing solvability to it when we have a relatively shallow depth, e.g. 2 or 3, but the feature had a negative coefficient when when we have relatively great depth, e.g. 12 or 13.

Feature Selection for Performance

After determining the best algorithm for searching Birds of a Feather puzzles and the best features based on predictability of solvability, we determined the best features for time-based performance. We investigated how different previously-described search features might yield improvements in node ordering so as to yield further time-based performance improvements. Whereas our best search algorithm, best-n DFS, ordered nodes only by *state flockability*, we now describe experiments that made use of different features for node ordering.

Having computed the aforementioned features on the solvability dataset, we decided to also test combined features for the best two and best three features. In order to accomplish this, we utilized the same process as used in our previous section Overall Feature Performance. That is, we perform a brute-force searching of the entire search space for the features. The output is then the best two or three features that we selected.

For each feature test, our nodes were ordered by the normalized value of each respective feature. In the case of multiple features, we used the logistic regression calculation by regressing values 1 and 0 for solvable and unsolvable states, respectively, onto various normalized features and normalized feature sets $\{x_1, x_2, \ldots, x_n\}$ in R in order to find the intercept and parameters $\{\theta_0, \theta_1, \ldots, \theta_n\}$ with the maximum likelihood to predict the solvability of a state using equation:

$$P(x) = \frac{1}{1 + e^{-(\theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n)}}$$

It should be noted that node ordering by $P(x)$ is equivalent to node ordering by $\theta_0 + \theta_1 x_1 + \ldots + \theta_n x_n$, as the logistic function is monotonic.

We next took all possible subsets of state features and tested them to determine root mean squared error in predicting solvability of a state. We found the best 2 features for solvability prediction to be *mobility per card* and *suit dominance*. The best 3 features were *mobility per card*, *suit dominance*, and *rank cluster count*. However, as shown in Table 3, search time performance degraded with the additional computation of these features.

The respective average search time per node of different features and feature combinations are shown in Table 3. Three tests were performed for each feature on 100,000 seeds in order to test variance.

Features compared amongst each other in our time-based testing allowed us to see whether more accurate prediction of solvability led to more rapid search for solutions. *State flockability* had by far the best performance when compared to our basic DFS, lowering average search time from 1,212 ms to 5.3 ms, a decrease of 99.56%. The *mobility per card* and *suit dominance* features produced the second and third best time decreases with 14 ms and 15 ms, respectively. Moreover, our best two features, *mobility per card* and *suit dominance*, and our best three features *mobility per card*, *suit dominance*, and *rank cluster count*, both underperformed in terms of computational time with respect to our

Features	**Time/Seed (ms)**
State flockability	5.3
Mobility	16.3
Minimum degree	24.5
Mobility per card	14
Mobility-flockability ratio	69
Rank cluster count	45
Rank deviation	23.4
Suit dominance	15
Flockability diameter	115
Best 2 features	29.5
Best 3 features	50

Table 3: Search time per puzzle seed for 100,000 seeds using different features for node ordering.

best feature *state flockability* with decreased average search times of 29.5 ms and 50 ms, respectively. We thus conclude that the greater accuracy of solvability prediction that can be achieved with combinations of features does not reduce node count sufficiently to offset the additional computational cost of computing these additional features.

Most Difficult Puzzle Seeds

We conclude with an examination of which puzzle seeds gave our best solver the greatest difficulty, maximizing node count and time. We begin by examining the unsolvable puzzle generated by seed 360,528, resulting in our maximum node count of 4,013,578:

5S	7D	QS	2C
4C	AD	8D	5C
QH	7C	TC	3C
8C	2D	9C	3D

Our first observation is that QH only flocks with QS which only has one other flocking possibility with 5S. Thus QS must come into the same row or column as QH via the 5S or TC initial positions. We can immediately eliminate QS-5S as a possibility as it leaves the queens as a separated flock. However, this does not alone imply unsolvability, as there could be some means by which the 5S travels to the TC position while all other cards are eliminated, allowing a penultimate QS-5S move downward. This is not the case, but the lack of solution here is not so straightforward as odd bird or separated flock checking.

We see a similar difficulty with our second most computationally expensive seed 731,678 with a node count of 3,757,599:

TS	AH	5C	QH
AS	AD	8D	QC
2S	3S	2H	3D
9S	TD	8S	9D

Here, the only two clubs, 5C and QC, are in the same role as the two queens of seed 360,528, and the QH is in the same role as the 5S. 5C and QC must flock, QC-QH results in immediate separated flocks, but there are many move sequences one might search out seeking to eliminate all other cards, bringing QH to the initial 8D position. These two puzzles suggest that one might use such analysis to set up a subgoal that would drive search, allowing a different type of pruning should that subgoal become impossible.

Surprisingly, seed 557,593, the most difficult *solvable* seed for our algorithm with a node count of 1,227,449, was not difficult for us to solve manually:

QH	4S	9H	8C
TC	JD	QS	9C
7C	9D	TH	7D
8D	2C	TD	7H

Our solution, "QS-JD QS-4S QH-QS 7C-8D 8C-9C 8C-TC 8C-7C 8C-2C 9H-QH 9H-8C TD-TH 7D-TD 7D-9D 7H-9H 7H-7D", begins by noting that spades are rare and the 4S only flocks with QS, so we initially flock QS over 4S via JD and flock QH over QS, reducing the number of suits to three. Finding it easier to visually reason about suit elimination, we then flock the clubs together and flock over the last club. From there, the solution is fairly straightforward and underconstrained. It seems that with this puzzle, as with our most computationally difficult *unsolvable* puzzles, the minimum degree cards suggests a goal-directed approach that would be beneficial to our heuristic search in such situations.

However, our heuristics make the common case fast, so one possible future line of work would be to create an ensemble of search algorithms working in parallel so that strategies such as these might find or disprove the existence of a solution faster.

Conclusions

In this work, we first sought to develop a highly efficient solver for Birds of a Feather puzzles, beginning with an evaluation of different search algorithms. Having experimented with various search algorithms, node ordering with various search features, and pruning according to odd-bird and separated flock properties, we first determined that our new search algorithm, best-n depth-first search performs the best, reducing basic DFS search time per puzzle seed from 1212 ms to 5.3 ms, a 99.56% decrease in search time. Algorithm 1, best-n depth-first search, organizing DFS by *cohort* and orders cohort children according to state flockability.

We next developed a solvability dataset and performed a study of a wide range of potential features for node ordering, employing exhaustive selection with respect to speed performance measurements. To our surprise, our initial single feature of state flockability proved best for node ordering.

Shifting our question toward that of correctly *predicting* solvability of states, we learned that state flockability is generally outperformed by the mobility per card feature, and that this superior prediction performance holds for all depths of search.

Finally, we took a closer look at the puzzle seeds which required the most computation from our best solver algorithm, gaining insight to common structures seen in both solvable and unsolvable puzzle seeds that are computationally expensive.

Given that (1) our average performance is greatly impacted by such computationally expensive outliers, and (2) common structural features appear in such outliers, we expect that future work on parallel subgoal-directed search algorithms could further improve upon what is already a considerable more efficient search algorithm for Birds of a Feather.

Acknowledgments

This work was supported, in part, by the Cross-Disciplinary Science Institute at Gettysburg College (X-SIG) and the Symposium on Educational Advances in Artificial Intelligence.

References

Beasley, J. 1985. *The ins and outs of peg solitaire*. Recreations in mathematics. Oxford University Press.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2009. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition.

Korf, R. E. 1993. Linear-space best-first search. *Artif. Intell.* 62(1):41–78.

Neller, T. W. 2016. AI education: Birds of a feather. *AI Matters* 2(4):7–8. Accessed 2018-08-07 via https://sigai.acm.org/static/aimatters/2-4/AIMatters-2-4-03-Neller.pdf.

Russell, S., and Norvig, P. 2009. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall, 3rd edition.