Optimal Solitaire Game Solutions using A* Search and Deadlock Analysis

Gerald Paul
Boston University
Boston, Massachusetts, USA
gerryp@bu.edu

Malte Helmert
University of Basel
Basel, Switzerland
malte.helmert@unibas.ch

Abstract

We propose and implement an efficient method for determining optimal solutions to such skill-based solitaire card games as Freecell and King Albert solitaire. We use A* search together with an admissible heuristic function that is based on analyzing a directed graph whose cycles represent deadlock situations in the game state. To the best of our knowledge, ours is the first algorithm that efficiently determines optimal solutions for Freecell games. We believe that the underlying ideas should be applicable not only to games but also to other classical planning problems which manifest deadlocks.

1 Introduction

Games have always been a fertile ground for advancements in computer science, operations research and artificial intelligence. Solitaire card games, and Freecell in particular, have been the subject of study in both the academic literature (Elyasaf, Hauptman, and Sipper 2011; 2012; Sipper and Elyasaf 2014; Long and Fox 2000; Bacchus 2001; Fox and Long 2001; Helmert 2003; Hoffmann 2005; Hoffmann and Nebel 2001; Hoffmann, Porteous, and Sebastia 2004; Morris, Tarassenko, and Kenward 2005; Pecora and Cesta 2003; Russell and Norvig 2003), where they are used as a benchmark for planning heuristics, and in popular literature (Fish 2015; Heineman 2015; FreeCell solutions 2015; Keller 2015; PySolFC 2015; Levin 2008; Van Noorden 2006; Mlot 2015).

Our work applies to *skill-based* solitaire games in which all cards are dealt face up. For these games, after the initial deal, there is no element of chance involved. Examples of such games include Freecell, King Albert, Bakers dozen, and Eight-off (Morehead and Mott-Smith 1983). Skill-based solitaire games are examples of *classical planning* problems (Ghallab, Nau, and Traverso 2004).

The Freecell solitaire game was introduced by Microsoft as a free desktop game in early versions of the Windows operating system. The rules of Freecell are described in Appendix A. While our examples and solution results are for Freecell, they apply to a large class of skill-based solitaire games.

We use Freecell because it is the most widely played and analyzed skill solitaire card game with free on-line, desktop and mobile versions of the game. Freecell games are denoted by the randomization seed that produces them in the Windows implementation of the game. Because the randomization algorithm for Freecell deals is public (Horne 2015), given a seed the random deals are reproducible, so comparisons can be made with other work. While Freecell differs in detail from other skill solitaire games, such concepts as foundation cells to which cards must ultimately be moved, and a tableau of columns of cards is common to many skill solitaire games. Freecell has been shown to be NP-hard (Helmert 2003) and thus provides a demanding test of our approach.

There are a number of Freecell computer solvers available which provide solutions to any Freecell deal (Elyasaf, Hauptman, and Sipper 2011; Fish 2015; Heineman 2015; Keller 2015; PySolFC 2015). However, we know of no work which provides provably *optimal* solutions to solitaire games. We consider a solution optimal if no other solution exists which requires a smaller number of moves.

One of the defining attributes of such skill-based games as Freecell is that *deadlocks*[1] are present and, in order to resolve the deadlocks, actions are required that do not contribute directly to reaching the goal state. Deadlock has long been recognized as a feature that makes finding optimal solutions to planning problems hard (Gupta and Nau 1992). When the state of the game is appropriately mapped to a directed graph, the deadlocks are represented by cycles of the graph.

A key insight of this work is that very strong admissible heuristic functions for Freecell can be constructed by analyzing these deadlock cycles. Graph analysis has been employed in analysis of problem complexity (Gupta and Nau 1992) and planning heuristics (Helmert 2004). One of the main contributions of this work is that we show how it can be used to optimally solve a highly popular class of puzzles that have so far defied optimal solution.

In the following sections, we review the A* algorithm, describe our approach, and presents results of our solver implementation. We conclude by discussing connections to recent research in classical planning, future research directions and open questions towards the end of this paper. While our algorithmic contributions and experimental evaluation are fo-

[1] In general, a deadlock situation exists when an action, A, cannot be taken until another action, B, is taken but action B, cannot be taken until action A is taken (and the generalization to circular waiting of multiple actions.

cused on Freecell, we believe that the key insights underlying the deadlock heuristic have much wider applicability within and outside of classical planning.

2 A* Search Algorithm

The A* search algorithm (Hart, Nilsson, and Raphael 1968) uses a best-first search and finds a least-cost path from a given initial state to the goal state. As A* traverses the state space, it builds up a tree of partial paths. The leaves of this tree (called the *open set*) are stored in a priority queue that orders the leaf states by a cost function:

$$f(n) = g(n) + h(n). \quad (1)$$

Here, $g(n)$ is the known cost of getting from the initial state to state n. $h(n)$ is a heuristic estimate of the cost to get from n to the goal state. For the algorithm to find the actual least cost path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the goal state. Roughly speaking, the closer the heuristic estimate is to the actual cost of reaching the goal state, the more efficient the algorithm.[2]

In our case, $g(n)$ is simply the number of moves that have been made to reach the state n and $h(n)$ is an estimate of the number of moves to reach the solution of the game from state n.

3 Freecell Heuristics for A*

The simplest non-trivial heuristic is $52 - m_f(n)$ where $m_f(n)$ is the number of cards in the foundation in state n. This estimate, however, is extremely optimistic because some cards are usually blocked from movement to the foundation. The simplest example of this is a column where a card of a given suit and rank is higher[3] in the column than a card of the same suit and greater rank. Until the greater rank card is moved to a temporary location in a free cell or another tableau cell, the lower rank card cannot be moved to the foundation. Then later, the greater rank card may be able to be moved to the foundation. Thus, a more robust heuristic is

$$h(n) = 52 - m_f(n) + m_e(n) \quad (2)$$

where $m_e(n)$ is an optimistic estimate of the number of moves to temporary locations that must be made to remove these blocking or deadlock situations.

There are more complicated deadlock situations than the example above. With the mapping of the game state to a directed graph described in the next section, we can associate all deadlock situations with cycles in the graph. The deadlock situations are removed when all cycles are eliminated; a cycle is eliminated when one or more edges of the cycle are removed. Now, the only way to remove an edge is to move a card and moving a card cannot remove more than one edge. Thus, the number of remaining moves must be at least as great as the number of moves to eliminate these cycles. For this reason, we take $m_e(n)$ to be an optimistic estimate of the number of edges which must be removed to eliminate all cycles. Note that this estimate may still not be an exact estimate of the number of remaining moves needed to win because the use of temporary locations is limited by the availability of open free cells, empty cells in the tableau, or a column to which the card can be moved to extend a cascade. Also note that we can use for $m_e(n)$ an estimate of the number of edges needed to remove a consistent *subset* of all cycles. This allows for performance tuning the implementation as discussed in Section 10.

[2]While not universally true (Holte 2010), the rule "more accurate heuristic = lower search effort" is generally a good approximation of reality.

[3]Throughout the paper, we use "higher" and "lower" to refer to the usual *visual* representation of card columns in solitaire games. For example, the "lowest" card in the leftmost column of Fig. 1 is 6♠. This is the only card in the column that may be moved directly. To move any other cards, the cards below them must first be moved out of the way.

4 Solitaire State to Directed Graph Mapping

We map the solitaire game layout to a directed graph as follows:

- We treat each card as a node of the graph.
- We create a directed edge from each card to the card of next lower rank of the same suit (e.g., from the 8♡ to the 7♡). We call these edges *dependency edges* because being able to move a card to the foundation depends on the card of next lower rank of the same suit being in the foundation. Dependency edges are permanent; they are never removed and are not affected when a card is moved. We define the *suit of a dependency edge* as the suit of the cards to which the edge is incident.
- We create a directed edge from each card in the tableau to the card below it in the tableau (if any). We denote these edges *blocking edges* because a card in the tableau is blocked from being moved to the foundation unless it is the exposed card (lowest card) in the tableau column. Blocking edges are removed and added when a move is made to reflect the new state of the game.

An example of dependency and blocking edges that are part of a cycle is shown in Fig. 1.

5 Cycle Determination

At first glance, the task of dealing with the cycles of the created by the mapping is daunting. For example, there are 26133 unique cycles in the graph created from mapping Freecell game #1 .

However, we can reduce the number of cycles to be considered by eliminating *redundant cycles*. A cycle c_1 is redundant if the set of blocking edges of any other cycle c_2 is a subset of the set of edges of c_1, in which case the removal of any edge in cycle c_2 results not only in the destruction of c_2 but also c_1.

Here we describe how to construct all non-redundant cycles. The approach depends on the fact that all dependency edges are always present. So from any card in a suit to any card of lower rank of the same suit we can always create a path that does not contain any blocking edges.

For conciseness, let us define a cycle that includes dependency edges of q different suits as a *q-suit cycle*. Now, first

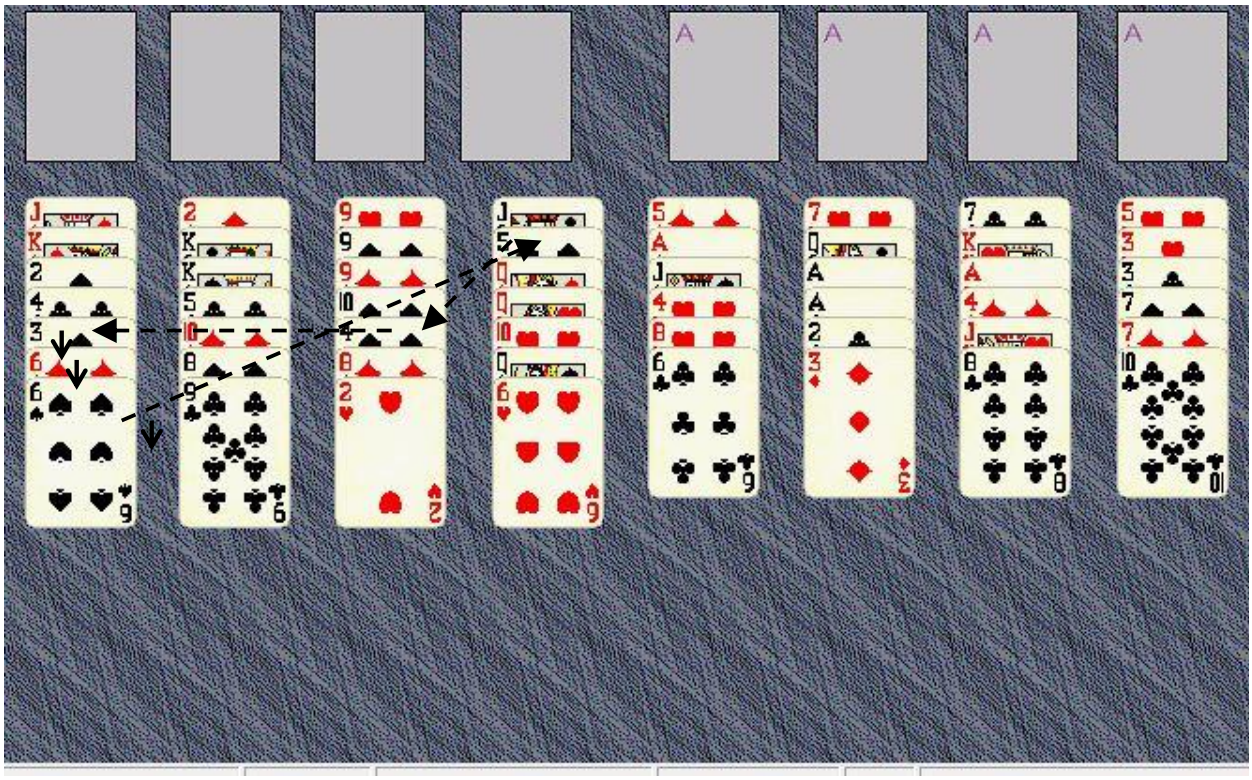

Figure 1: Initial state of Microsoft Freecell game #1. A cycle consisting of blocking edges (solid arrows) $3\spadesuit \rightarrow 6\diamondsuit, 6\diamondsuit \rightarrow 6\spadesuit$ and dependency edges (dashed lines) $6\spadesuit \rightarrow 5\spadesuit, 5\spadesuit \rightarrow 4\spadesuit, 4\spadesuit \rightarrow 3\spadesuit$ is shown.

consider cycles in which all dependency edges are of the same suit (1-suit cycles), an example of which is shown in Fig. 2. The blocking edges in the cycle in Fig. 2(a) are a subset of the blocking edges in the cycle in Fig. 2(b). So the cycle in Fig. 2(b) is a redundant cycle. We can infer the rule that if all dependency edges of a cycle are of the same suit, we need only consider cycles in which the dependency edges of the same suit in the cycle are consecutive (i.e., not interrupted by blocking edges). This rule implies that all blocking edges in a single-suit cycle are in the same tableau column.

Now consider cycles in which the dependency edges include cards of two suits (2-suit cycles), an example of which is shown in Fig. 3. Again, the cycle in Fig. 3(b) is redundant because blocking edges in the cycle in Fig. 3(a) are a subset of the blocking edges of the cycle in Fig. 3(b). Generalizing to cycles containing dependency edges of any number of suits, we can infer that

- if the dependency edges of any suit in the cycle are not consecutive, (i.e., are interrupted by blocking edges) the cycle is redundant, and that
- for a cycle containing dependency edges of q suits, the blocking edges of non-redundant cycles are contained in at most q tableau columns.

Based on the above, we can construct non-redundant q-suit cycles, $(q = 1, 2, 3, 4)$, by considering $O(t^q)$ combinations of the tableau columns, where t is the number of tableau columns (8 for Freecell). Pseudo code for identification of 1-suit and 2-suit cycles is presented in Appendix B.

6 Relevant Cycle Edges

In considering edges for removal from a cycle, not all edges must be considered. Clearly, only blocking edges (as opposed to dependency edges) must be considered, since dependency edges are never removed.

Also, in solitaire, cards higher in a column than a given card cannot be moved before the given card is moved. Thus, as seen in the example in Fig. 2(a), while the edge $2\diamondsuit \rightarrow A\clubsuit$ is part of the cycle shown, if the edge $A\clubsuit \rightarrow 9\spadesuit$ is removed, which must be done before the edge $2\diamondsuit \rightarrow A\clubsuit$ can be removed, the cycle no longer exists. Thus, there is no reason to consider the edge $2\diamondsuit \rightarrow A\clubsuit$ as a candidate for removal during the calculation of edges required to be removed. We denote edges that must be considered as candidates for removal as *relevant cycle edges* and edges that need not be considered *irrelevant cycle edges*.

7 Duplicate Cycles

Since we are only concerned with the number of edges which must be removed to eliminate cycles, we can remove from consideration cycles which are duplicates of other cycles. We consider cycles to be duplicates if the set of rel-

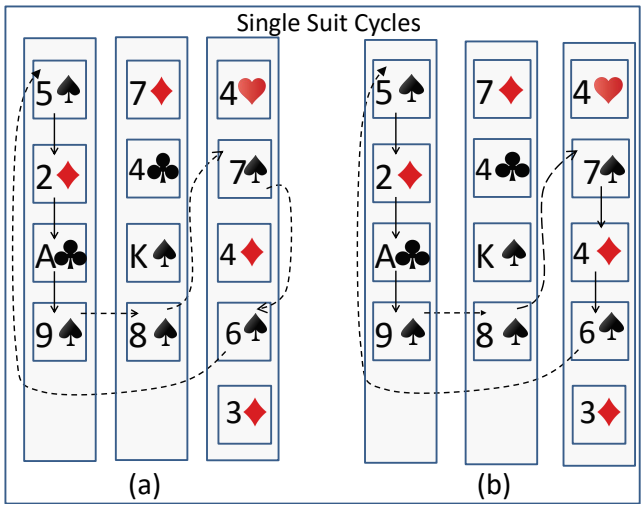

Figure 2: Fragment of tableau illustrating (a) single suit cycle in which all blocking edges are in a single column and (b) redundant single suit cycle consisting of blocking edges (in two columns) that are a superset of blocking edges in (a).

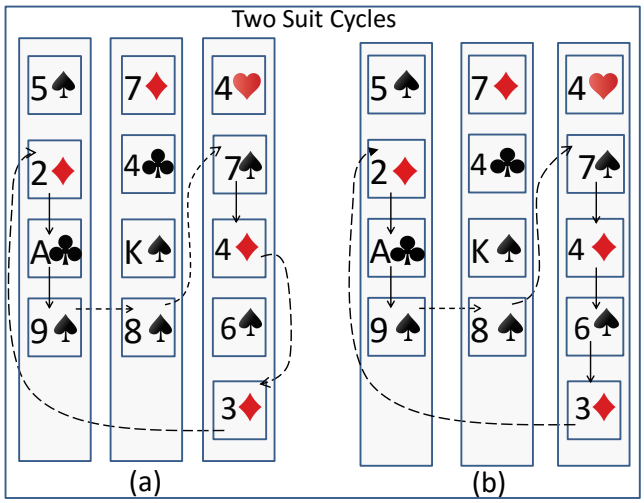

Figure 3: Fragment of tableau illustrating (a) 2-suit cycle in which all blocking edges are in a two columns and (b) redundant 2-suit cycle consisting of blocking edges (also in two columns) that are a superset of blocking edges in (a).

evant edges in the cycles are identical, independent of the order, and eliminate them from consideration.

8 Determination of the Minimum Number of Edges that Must be Removed to Eliminate Cycles

By eliminating redundant cycles, duplicate cycles and irrelevant edges of cycles, the complexity of determining the minimum number of edges that must be removed to eliminate cycles, m_e, can be reduced significantly. For example, we must actually only consider a total of 87 cycles (12 1-suit, 39 2-suit, 34 3-suit, and 2 4-suit cycles) for removal in the initial state of Freecell game #1, compared to the 26133 cycles present if redundant and duplicate cycles are included.

We use a brute-force, depth-first exhaustive search of all combinations of edge removals that eliminate all cycles. Because only one edge of a cycle must be removed to eliminate the cycle, the worst case number of combinations of removed edges that we must consider is:

$$C = \prod_{i=1}^{m_c} e_i, \tag{3}$$

where m_c is the number of cycles and e_i is the number of relevant edges in cycle i.

In practice, the number of combinations actually considered can be reduced, in some cases by an order of magnitude, by removing edges in decreasing order of the number of other cycles in which an edge is contained. With this ordering many cycles are eliminated early in the calculation.

9 Implementation

Our solver is implemented in C++. In addition to eliminating redundant cycles, duplicate cycles and non-relevant edges, we improve performance with:

- A transposition table with Zobrist hashing to eliminate duplicate states during the A* search (Akagi, Kishimoto, and Fukunaga 2010; Zobrist 1990).
- The priority queue used by A* implemented as a series of buckets (as opposed to a binary heap), providing $O(1)$ queue performance (Paul 2007; Burns et al. 2012). This is possible because the costs (estimated solution lengths) are integers and in a relatively small range ($\approx$ 60–100).
- Use of a hash table to detect duplicate cycles.
- Incremental cycle determination. We identify all cycles in the initial configuration. After that, we incrementally identify cycles removed and added as a result of a move, only considering the card(s) moved.

10 Solver Results

Our test cases were games 1–5000 of Microsoft Freecell. We ran all tests on an Intel core i3 4160 processor running at 3.60 GHz with 8 GB of memory. The program working set is $\approx$ 2 GB for most games. Results for games 1–10 are shown in Table 1.

Before discussing the results, it is necessary to discuss the trade-off between the accuracy of our heuristic and the computer resources to achieve that accuracy. Our work can be thought of as providing a family of heuristic functions, $h_q(n)$, where $q = 1, 2, 3, 4$ is the maximum number of different suits of the dependency edges in the cycles we consider. Now, the processing time per state explored to determine cycles increases with q. The number of states which must be explored, however, decreases with q because the heuristic becomes more accurate with increasing q and the heuristic more effectively guides the search. Reducing the

	initial state			solution		
Freecell Game #	1-suit cycles	2-suit cycles	length estimate	length	time (sec)	states searched
1	12	39	73	82	30.8	567699
2	13	5	68	73	1.9	101186
3	15	8	70	70	0.6	20499
4	37	34	72	79	31.0	1220026
5	16	46	78	85	122.0	3687136
6	12	24	73	75	1.1	31912
7	13	39	72	76	1.7	74369
8	8	58	70	74	13.2	367784
9	19	25	77	81	2.0	77990
10	16	18	73	80	7.7	315643

Table 1: Results of our program for the Freecell games 1-10.

number of states which must be explored is important because it also reduces the amount of memory A* requires and memory is often the limiting factor in A* implementations.

Varying q, we find that, while using cycles containing more than 2 suits reduces the number of states needed for a solution, the time per state is increased so much that the overall average time per solution is increased. For this reason, we did not identify or use cycles containing greater than 2 suits in our testing.

We found that across the 5000 games used for testing

- Optimal solutions have been found for all games tested.
- As expected, the initial estimated length is never greater than the actual optimal length – a requirement for the search to find the optimal solution.
- The initial estimated length and the actual optimal length are relatively close – in some cases (e.g., game 3) identical, indicating that the heuristic strongly guides the search.
- There are typically $\lessapprox 20$ initial 1-suit cycles and $\lessapprox 100$ 2-suit cycles.
- CPU processing time is dominated by the tasks of finding cycles and determining which edges must be removed to eliminate all cycles.
- The shortest, average and longest solution lengths were 64, 77 and 93 moves, respectively.
- The shortest, average and longest processing times were 0.4, 39.9 and 6579 seconds, respectively.

11 Solitaire and Blocks World

By mapping game states to directed graphs and using the number of edges required to remove cycles in the graph as input to the A* heuristic function, we develop, for the first time, an effective method for finding optimal solutions to skill-based solitaire games. These games are characterized by the presence of deadlock situations in which the goal state requires objects to be ordered in a specified way but constraints exist on the order in which actions can be taken. The presence of deadlocks in these games makes them hard to solve, and in particular hard to solve optimally.

This is reminiscent of the classic blocks world domain, where nonoptimal solutions can be generated easily, but computing optimal solutions is an NP-equivalent problem due to the existence of deadlocks (Gupta and Nau 1992; Slaney and Thiébaux 2001). Indeed, the Freecell heuristics described in this paper can be understood as a two-stage relaxation. Firstly, relax the Freecell game into a blocks world task. Secondly, compute an admissible heuristic for this blocks world task.

In more detail, we first relax the Freecell game by treating it as a blocks world task where the cards forming each tableau column or foundation pile are reinterpreted as a tower of blocks, and cards in free cells are reinterpreted as individual blocks lying on the table. This simplifies the problem (and hence leads to an admissible heuristic rather than a perfect distance estimate) because blocks world, unlike Freecell, has unlimited table positions. Interestingly, removing the constraint on table positions is the *only* relevant way in which this transformation simplifies the problem: while the Freecell rules impose a number of constraints regarding the movement of cards, none of these constraints affect the optimal solution length in the presence of unlimited table positions. In other words, Freecell with unlimited table positions (or unlimited free cells) always has exactly the same optimal solution length as the corresponding blocks world task.[4]

The second relaxation we apply in our experiments is to abstain from covering all deadlocks of the problem graph. Recall that h_q ($1 \leq q \leq 4$) is the variant of our heuristic that only considers q'-suit cycles with $q' \leq q$. The most

[4]To see this, observe that with unlimited space there is never an incentive in Freecell to move a card onto another card except for its final move to foundations. This is equivalent to the observation that in blocks world, there is never an incentive to move a block onto another block except to move it into its final position. The challenge, in both cases, is to minimize the number of moves of cards/blocks onto the table that cannot yet be moved directly into their final position.

powerful of these heuristics, h_4, includes all relevant cycles and hence amounts to solving the blocks world relaxation of the Freecell game perfectly. In our experiments, this level of heuristic accuracy turned out not to be beneficial due to the high computational effort for each state evaluation, with h_2 providing the best balance between heuristic accuracy and computational effort per state.

12 Implications for Domain-Independent Planning

Looking beyond solitaire games and blocks world, are there wider implications of our work for domain-independent planning? We believe that this is the case: that deadlocks are a phenomenon that occurs in a much wider range of domains than Freecell games or blocks world tasks, and that heuristic functions based on covering deadlocks are a promising direction for a wide range of planning domains.

For example, deadlocks of essentially the same form as in the blocks world domain are the major source of hardness in the Logistics domain and the only source of hardness in the Miconic-STRIPS and Miconic-SimpleADL domains (Helmert 2001). Many other planning domains with a "transportation" component share this problem aspect, though often mixed with other aspects. Deadlock covering problems also occur at the computational core of many optimization problems outside of planning, such as many of the *implicit hitting set* problems identified by Chandrasekaran et al. (2011).[5]

Finally, a similar form of deadlocks (a set of actions cyclically supporting each other's preconditions without being ultimately supported by effect/precondition links from the current state) is the major source of inaccuracy in *flow heuristics* that have recently attracted much attention in planning (van den Briel et al. 2007; Bonet 2013; Bonet and van den Briel 2014; Pommerening et al. 2014). A better understanding of the role of dependency deadlocks in classical planning tasks could go a long way towards overcoming the limitations of these heuristics.

References

Akagi, Y.; Kishimoto, A.; and Fukunaga, A. 2010. On transposition tables for single-agent search and planning: Summary of results. In Felner, A., and Sturtevant, N., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, 2–9. AAAI Press.

Bacchus, F. 2001. The AIPS'00 planning competition. *AI Magazine* 22(3):47–56.

Bonet, B., and van den Briel, M. 2014. Flow-based heuristics for optimal planning: Landmarks and merges. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 47–55. AAAI Press.

[5] See also the recent paper by Slaney (2014) for deeper connections between blocks world, implicit hitting sets, and combinatorial optimization in general.

Bonet, B. 2013. An admissible heuristic for SAS$^+$ planning obtained from the state equation. In Rossi, F., ed., *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI 2013)*, 2268–2274.

Burns, E.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing fast heuristic search code. In Borrajo, D.; Felner, A.; Korf, R.; Likhachev, M.; Linares López, C.; Ruml, W.; and Sturtevant, N., eds., *Proceedings of the Fifth Annual Symposium on Combinatorial Search (SoCS 2012)*, 25–32. AAAI Press.

Chandrasekaran, K.; Karp, R.; Moreno-Centeno, E.; and Vempala, S. 2011. Algorithms for implicit hitting set problems. In Randall, D., ed., *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithm (SODA 2011)*, 614–629. SIAM.

Elyasaf, A.; Hauptman, A.; and Sipper, M. 2011. GA-FreeCell: evolving solvers for the game of FreeCell. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation (GECCO 2011)*, 1931–1938.

Elyasaf, A.; Hauptman, A.; and Sipper, M. 2012. Evolutionary design of FreeCell solvers. *IEEE Transactions on Computational Intelligence and AI in Games* 4(4):270–281.

Fish, S. 2015. Freecell solver. `http://fc-solve.shlomifish.org/`. Retrieved 11/9/2015.

Fox, M., and Long, D. 2001. Hybrid STAN: Identifying and managing combinatorial optimisation subproblems in planning. In Nebel, B., ed., *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI 2001)*, 445–452. Morgan Kaufmann.

FreeCell solutions. 2015. FreeCell solutions to 1000000 games. `http://freecellgamesolutions.com/`. Retrieved 11/9/2015.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Gupta, N., and Nau, D. S. 1992. On the complexity of blocks-world planning. *Artificial Intelligence* 56(2–3):223–254.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2):100–107.

Heineman, G. T. 2015. Algorithm to solve FreeCell solitaire game. `http://broadcast.oreilly.com/2009/01/january-column-graph-algorithm.html`. Retrieved 11/9/2015.

Helmert, M. 2001. On the complexity of planning in transportation domains. In Cesta, A., and Borrajo, D., eds., *Proceedings of the Sixth European Conference on Planning (ECP 2001)*, 120–126. AAAI Press.

Helmert, M. 2003. Complexity results for standard benchmark domains in planning. *Artificial Intelligence* 143(2):219–262.

Helmert, M. 2004. A planning heuristic based on causal graph analysis. In Zilberstein, S.; Koehler, J.; and Koenig, S., eds., *Proceedings of the Fourteenth International Confer-*

ence on Automated Planning and Scheduling (ICAPS 2004), 161–170. AAAI Press.

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

Hoffmann, J.; Porteous, J.; and Sebastia, L. 2004. Ordered landmarks in planning. *Journal of Artificial Intelligence Research* 22:215–278.

Hoffmann, J. 2005. Where 'ignoring delete lists' works: Local search topology in planning benchmarks. *Journal of Artificial Intelligence Research* 24:685–758.

Holte, R. C. 2010. Common misconceptions concerning heuristic search. In Felner, A., and Sturtevant, N., eds., *Proceedings of the Third Annual Symposium on Combinatorial Search (SoCS 2010)*, 46–51. AAAI Press.

Horne, J. 2015. Description of microsoft FreeCell shuffle algorithm. `http://www.solitairelaboratory.com/mshuffle.txt`. Retrieved 11/9/2015.

Keller, M. 2015. Solitaire laboratory. `http://solitairelaboratory.com/index.html`. Retrieved 11/9/2015.

Levin, J. 2008. Solitaire-y confinement: Why we can't stop playing a computerized card game. *Slate* May 16, 2008.

Long, D., and Fox, M. 2000. Automatic synthesis and use of generic types in planning. In Chien, S.; Kambhampati, S.; and Knoblock, C. A., eds., *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS 2000)*, 196–205. AAAI Press.

Mlot, S. 2015. Microsoft tournament celebrates 25 years of solitaire. `http://www.pcmag.com/article2/0,2817,2484370,00.asp`. Published May 19, 2015; retrieved 11/9/2015.

Morehead, A. H., and Mott-Smith, G. 1983. *The Complete Book of Solitaire and Patience Games*. Bantam.

Morris, R.; Tarassenko, L.; and Kenward, M. 2005. *Cognitive Systems – Information Processing Meets Brain Science*. Elsevier.

Paul, G. 2007. A complexity $o(1)$ priority queue for event driven molecular dynamics simulations. *Journal of Computational Physics* 221(2):615–625.

Pecora, F., and Cesta, A. 2003. The role of different solvers in planning and scheduling integration. In *Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence (AI*IA 2003)*, 362–374.

Pommerening, F.; Röger, G.; Helmert, M.; and Bonet, B. 2014. LP-based heuristics for cost-optimal planning. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2014)*, 226–234. AAAI Press.

PySolFC. 2015. PySolFC: a Python solitaire game collection. `http://pysolfc.sourceforge.net/`. Retrieved 11/9/2015.

Russell, S., and Norvig, P. 2003. *Artificial Intelligence — A Modern Approach*. Prentice Hall.

Sipper, M., and Elyasaf, A. 2014. Lunch isn't free, but cells are: Evolving FreeCell players. *SIGEvolution newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation* 6(3–4):2–10.

Slaney, J., and Thiébaux, S. 2001. Blocks World revisited. *Artificial Intelligence* 125(1–2):119–153.

Slaney, J. 2014. Set-theoretic duality: A fundamental feature of combinatorial optimisation. In Schaub, T.; Friedrich, G.; and O'Sullivan, B., eds., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI 2014)*, 843–848. IOS Press.

van den Briel, M.; Benton, J.; Kambhampati, S.; and Vossen, T. 2007. An LP-based heuristic for optimal planning. In Bessiere, C., ed., *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, 651–665. Springer-Verlag.

Van Noorden, R. 2006. Computer games could save your brain. *Nature* news item published 24 July 2006.

Wikipedia. 2015. FreeCell. `https://en.wikipedia.org/wiki/FreeCell`. Retrieved 11/9/2015.

Zobrist, A. L. 1990. A new hashing method with application for game playing. *ICCA Journal* 13(2):69–73.

A Freecell Rules

The following Freecell rules are taken from Wikipedia (2015).

Construction and layout: One standard 52-card deck is used. There are four open cells and four open foundations. Cards are dealt face-up into eight cascades, four of which comprise seven cards and four of which comprise six.

Building during play: The top card of each cascade begins a tableau. Tableaux must be built down by alternating colors. Foundations are built up by suit.

Moves: Any cell card or top card of any cascade may be moved to build on a tableau, or moved to an empty cell, an empty cascade, or its foundation. Complete or partial tableaus may be moved to build on existing tableaus, or moved to empty cascades, by recursively placing and removing cards through intermediate locations.

Victory: The game is won after all cards are moved to their foundation piles.

B Cycle Identification Algorithm

Figures 4 and 5 contain pseudo code for determining 1-suit and 2-suit cycles, respectively.

Note that there is no need to explicitly follow the dependency edges from a card in one column to a card of the same suit in another column since we are assured that there is always a chain of dependency edges from a card in a given suit to a card of lower rank in that suit.

Extension to 3-suit and 4-suit cycles is straightforward; paths of dependency edges to one or two additional columns, respectively, must be identified before returning to the initial column.

```
/* identify 1-suit cycles*/

for (all tableau columns,c)
{
   for (all cards, cardX,  in column c)
   {
      for (all cards,cardY, below cardX)
      {
         if (cardY suit != cardX suit || cardY rank > cardX rank)
            continue;   /* need same suit, lower rank than X*/

         /* cycle found */
         store cycle;
      }
   }
}
```

Figure 4: Pseudo code illustrating the algorithm to identify 1-suit cycles.

```
/* identify 2-suit cycles*/

for (all tableau columns, c1)
{
    for (all cards, cardX1, in column c1)
    {
        for (all tableau columns, c2)
        {
            for (all cards, cardY2 in column c2)
            {
                if (cardY2 suit != card X1 suit || cardY2 rank > cardX1 rank )
                    continue;    /* need same suit, lower rank than X1*/

                for (all cards, cardY1, above cardY2 in column c2)
                {
                    if (cardY1 suit == card Y2 suit)/* need different suit */
                        continue;

                    for (all cards, cardX2, below cardX1 in column c1)
                    {
                        if (cardX2 suit != cardY1 suit || cardX2 rank > cardY1 rank)
                            continue;    /* need same suit, lower rank than Y1*/

                        /* cycle found */
                        store cycle;
                    }
                }
            }
        }
    }
}
```

Figure 5: Pseudo code illustrating the algorithm to identify 2-suit cycles.