SEARCHING SOLITAIRE IN REAL TIME

Ronald Bjarnason *Prasad Tadepalli* *Alan Fern*[1]

Corvallis, USA

ABSTRACT

This article presents a new real-time heuristic search method for planning problems with distinct stages. Our multistage nested rollout algorithm allows the user to apply separate heuristics at each stage of the search process and tune the search magnitude for each stage. We propose a search-tree compression that reveals a new state representation for the games of Klondike Solitaire and Thoughtful Solitaire, a version of Klondike Solitaire in which the location of all cards is known. Moreover, we present a Thoughtful Solitaire solver based on these methods that can determine over 80% of Thoughtful Solitaire games in less than 4 seconds. Finally, we demonstrate empirically that no less than 82% and no more than 91.44% of Klondike Solitaire games have winning solutions, leaving less than 10% of games unresolved.

1. INTRODUCTION

Many AI problems from computer game playing to industrial process planning can be formulated as search problems, where the goal is to find a sequence of actions from an initial state to a goal state. In real-time search, one needs to commit to an action in a reasonable amount of time, i.e., before searching a significant fraction of the state space. Heuristic search methods belong to the most successful methods in solving planning-and-search problems (Bonet and Geffner, 2001) and are adapted to real-time search (Korf, 1990). An effective heuristic can help focus the search along fruitful directions. However, simply choosing an action greedily according to one-step lookahead is usually not effective because it leads to dead ends and local optima. Tuning of the heuristic via reinforcement learning is one way to circumvent this problem (Barto, Bradtke, and Singh, 1995; Korf, 1990). However, to be successful in large domains, this requires approximating the heuristic function (Tesauro, 1995). The errors in the approximated function ultimately limit the effectiveness of a greedy action choice.

In this article, we explore the use of *search* to amplify the effectiveness of the heuristic function in action selection. Multi-step depth-first lookahead is a standard search method effectively used in games. Recently, it has been found that rollouts and nested rollouts also amplify the effectiveness of a search heuristic in some domains (Tesauro and Galperin, 1996; Yan *et al.*, 2005). We call action selection using a greedy one-step lookahead, a level-0 rollout. A level n rollout considers all possible actions in the current state followed by level-$(n-1)$ rollouts until the end of the game (a win or a dead end). It then chooses the action that led to the best possible result.

Due to the inherent complexity of search spaces, multiple heuristics are often useful in different regions of the state space (Baxter, Tridgell, and Weaver, 2000; Chang, Givan, and Chong, 2004; Yoon, Fern, and Givan, 2005). In this article we introduce *multistage nested rollouts*, where different heuristics are used in different stages of the search space, and are combined with nested rollouts of different degrees of nesting. Multistage nested rollouts enable tight control of search magnitude at each stage in the search tree. Search can be increased in difficult stages, and conserved in simple stages. We demonstrate the effectiveness of this algorithm by creating a real-time solver for the game of Thoughtful Solitaire, a version of Klondike Solitaire where the location of all cards is known.

Klondike Solitaire has become an almost ubiquitous computer application, available to hundreds of millions of users worldwide on all major operating systems, yet theoreticians have struggled with this game, referring to the

[1]School of EECS, Oregon State University, Corvallis, OR, USA. Email:{ronny,tadepall,afern}@eecs.oregonstate.edu

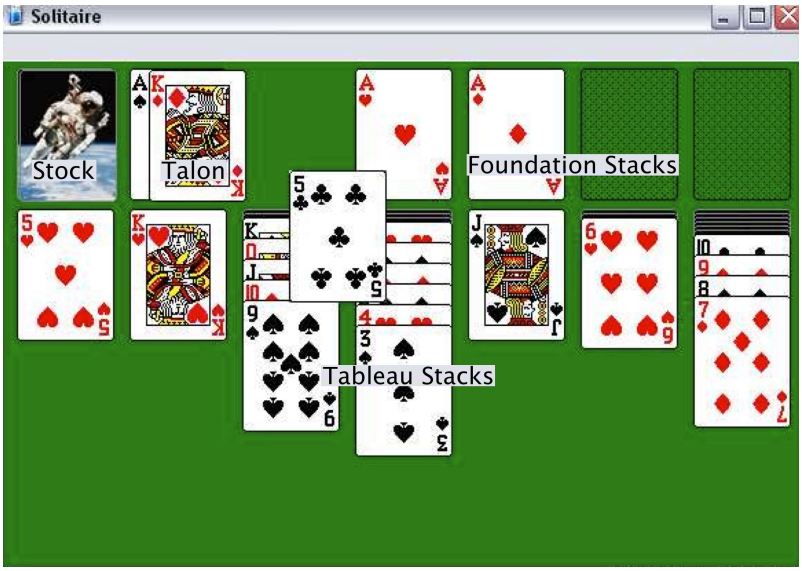

Figure 1: A screenshot of Klondike Solitaire. Each card stack is labelled with one of four types: the Talon, the Stock, the 4 Foundation Stacks, and the 7 Tableau Stacks.

inability to calculate the odds of winning a randomly dealt game as "one of the embarrassments of applied mathematics" (Yan *et al.*, 2005). While real-time solvers exist for similar games (Fish, 2005), no such solver exists for Klondike Solitaire. Independent developers are working to remedy this problem through various methods (Behnke, 2007; Kulow, 2006; Doug, 2007; Cronin, 2007), but none have yet fully integrated a real-time solver for the game of Klondike.

At this time, the best published result (in any Klondike variant) is by Yan *et al.* (2005). Using the slightly more stricter version of Thoughtful Solitaire, they are able to solve up to 70% of the games using level-3 rollouts in an average of 105 minutes per game. We empirically demonstrate that no less than 82% and no more than 91.44% of instances of Thoughtful Solitaire (and therefore Klondike Solitaire) have winning solutions and that over 80% of Thoughtful Solitaire games can be determined in less than 4 seconds, providing the foundation for a Thoughtful Solitaire game with real-time search capability. We provide such a game for free download at Bjarnason (2007).

What makes our search especially effective in Solitaire is the use of a compressed search tree, which combines sequences of individual moves into longer macros. A new representation for Thoughtful Solitaire and Klondike Solitaire based on this compressed search tree is described in Section 2. Section 3 describes our multistage nested rollouts algorithm. In Section 4, we describe additional domain-specific improvements to the search mechanism, including pruning methods and additional heuristics. Section 5 presents the results, and Section 6 concludes with discussion.

2. KLONDIKE SOLITAIRE AND VARIANTS

The following definitions apply to the game of Klondike Solitaire and its variants. A screenshot of a game of Klondike Solitaire can be seen in Figure 1.

Suit : ♢ (diamonds) and ♡ (hearts) are red. ♣ (clubs) and ♠ (spades) are black.

Rank : Each suit is composed of 13 cards, ranked (in ascending order): Ace, 2, 3, 4, ..., 10, Jack, Queen, King. For our value functions, we also refer to a 0-based **rank value**: (**A**=0, **2**=1, ..., **J**=10, **Q**=11, **K**=12).

Tableau or *build* or *base* stacks: seven stacks to build down card sequences in descending order and alternating colour. A card x can **block** another card y if x is above y in a Tableau stack and x is not resting on a face up card. Each card (except for Kings) has two **tableau build cards**, which are the two cards of opposite colour and next higher rank that a card can be placed on to build down the Tableau stacks.

Foundation or *suit* stacks: one for each suit (♢, ♣, ♡, ♠). The goal of the game is to build all 52 cards into the Foundation stacks in ascending order. Each card (except for Aces) has a **foundation build card** which is the suited card of preceding rank that a card can be placed on to build up the Foundation.

Stock or *deck*: holds face-down cards, which can be transferred to the Talon three at a time.

Talon or *discard* or *waste*: holds face-up cards transferred from the Stock. The top-most card on the Talon can be played to the Foundation or Tableau. When the Stock is emptied, the Talon can be recycled by placing it as a unit face down onto the Stock.

Some rules vary with versions of Klondike Solitaire. All methods and results in this article apply to those versions of Klondike Solitaire with three card Stock turns with unlimited deals, allowing movement of partial Tableau stacks and Foundation cards to the Tableau. Results reported from Yan *et al.* (2005) apply to a version that disallows movement of partial Tableau stacks.

2.1 Thoughtful Solitaire

The rules and structure of Thoughtful Solitaire are identical to those of Klondike Solitaire, with the exception that the identity and location of all 52 cards is known at all times. Because of this, the strategy of Thoughtful Solitaire differs from typical Klondike Solitaire play, with much more emphasis placed on deterministic planning.

2.2 A New State Representation: K^+ Solitaire

In any given Klondike Solitaire state, as few as one-third, or as many as two-thirds of the cards in the Talon and Stock are reachable through repeated play of the *Turn Stock* action, but only the card at the top of the Talon is playable. We introduce a new state representation that makes all *reachable* Stock and Talon cards immediately playable by combining states linked by *Turn Stock* actions. We call this new representation K^+ Solitaire. The action space of the compressed K^+ node is the union of the actions available from the uncompressed nodes, minus the eliminated *Turn Stock* actions. Figure 2 illustrates this tree compression. The new representation of the state-space can also be thought of as adding several macro-actions, where each macro consists of a series of *Turn Stock* actions followed by a *Play* action.

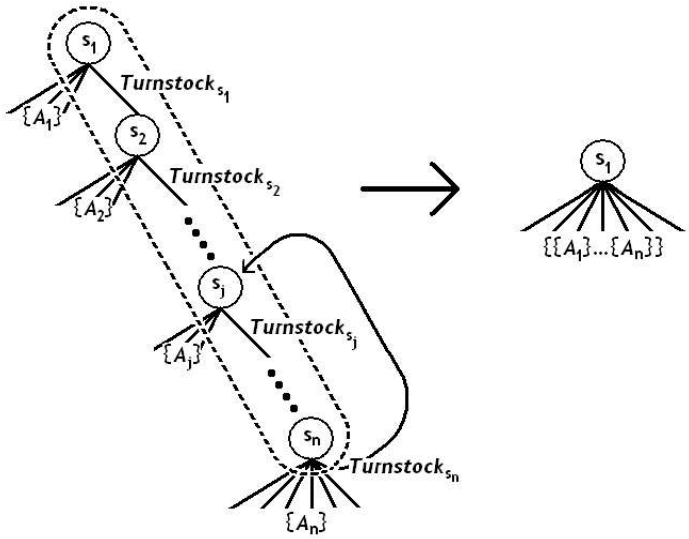

Figure 2: An illustration of the search-tree compression resulting from the elimination of the Stock. The search tree for Klondike Solitaire is compressed by eliminating each of the *Turn Stock* actions.

This new representation is graphically displayed in a Solitaire game we have made available at Bjarnason (2007). The face-up Talon is fanned out on the table, with all reachable cards highlighted for immediate play. A screenshot

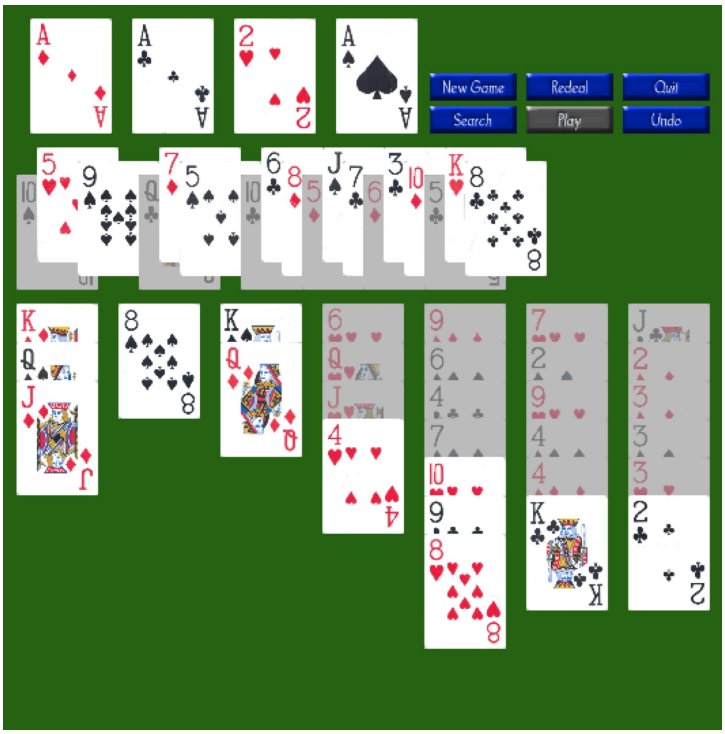

Figure 3: A graphical display of K^+ Solitaire.

of this display can be seen in Figure 3. Essential game play using the K^+ representation is not altered. All solutions found in the compressed search tree are applicable to Klondike Solitaire and Thoughtful Solitaire.

3. TRAVERSING THE SEARCH TREE

A deep search tree with a non-trivial branching factor prohibits brute-force search methods from being effective in Klondike Solitaire and its variants. Heuristic search methods fare well in such domains if a reasonable heuristic can be applied. Even with a good heuristic, search remains expensive. Yan *et al.* (2005) used a complex heuristic based on published best practices in playing Klondike Solitaire. In Table 2(b), we see a heuristic that on its own won 13% of games, with an improved performance resulting from the nested rollouts. Our search method is motivated by two factors. First, it appears that different stages of the game warrant different heuristics for guiding the search. Second, all stages may not need deeply nested searches. By tuning the level of rollout search for each stage, resources can be allocated to difficult stages and conserved in simple stages, with an appropriate heuristic applied to each stage.

3.1 Nested and Staged Rollouts

Rollouts were first described by Tesauro and Galperin (1996) as an effective method for amplifying the effects of a search heuristic. Yan *et al.* (2005) adapted rollouts through nested recursive calls to the basic method. Each degree of nesting increases the search time exponentially, but further amplifies the effects of the base heuristic. Given a heuristic function that maps each state to a real number, a greedy policy with respect to this heuristic chooses the action that results in a state that has the highest value. We refer to this as a level-0 rollout policy. Given a level-n rollout policy π, a level-$(n+1)$ rollout policy is constructed as follows: the value of each action a in a state s is estimated by choosing a, and then following a level-n rollout policy thereafter for a predetermined length (or until termination). After this step, the action that leads to the highest backed-up value is returned. By this definition, a level-1 rollout policy is analogous to the traditional rollout method described in Tesauro and

```
procedure rollout(s, h)
1  while s is not a dead-end or a win
2     a' = argmax_a (greedy( result(s,a), h)
3     s = result(s,a')
4  return h(s)

procedure greedy(s, h)
1  while s is not a dead-end or a win
2    a' = argmax_a ( h(result(s,a) )
3    s = result(s,a')
4  return h(s)
```

(a)

```
procedure nested-rollout(s, h, n)
1  while s is not a dead-end or a win
2    if n is 0
3       a' = argmax_a (greedy(result(s,a),h))
4    else
3      a' = argmax_a (nested-rollout(result(s,a), h, n-1))
4    s = result(s, a')
5  return h(s)
```

(b)

```
procedure multistage-nested-rollout (s, h_0, h_1, ... , h_z, n_0, n_1, ... , n_z)
 1  while s is not a dead-end or a win
 2    if n_0 is -1
 3      return h_0(s,a)
 4    else
 5      val = max_a (multistage-nested-rollout(result(s,a), h_0, ... h_z, n_0-1, ... n_z)
 6        and let a' be the maximizing action
 7      if h_0(s) > val and h_0 is not the last heuristic
 8        a' = argmax_a (multistage-nested-rollout(s, h_1, ... h_z, n_1, ... n_z)
 9    s = result(s, a')
10  return h_0(s)
```

(c)

Figure 4: Basic descriptions of (a) standard rollouts, (b) nested rollouts, and (c) multistage nested rollouts. Each *h* represents a heuristic that can evaluate a state, *s* to a real value with h(*s*). Each *n* is a level of nesting, associated with a heuristic. At each level of nesting, the algorithm attempts to determine which action, *a*, should be taken.

Galperin (1996). The greedy policy and standard rollout methods are described in Figure 4(a) and graphically depicted in Figure 5(a) and Figure 6, respectively. Figure 4(b) describes the nested rollout algorithm.

In many domains, it is unrealistic to assume a single heuristic will perform well across all stages of the search tree (Baxter *et al.*, 2000; Chang *et al.*, 2004). Increasing the nesting will eventually overcome the weaknesses of using a single heuristic, but only at the expense of increased search throughout the entire search tree, even in stages where additional search may not be warranted. The ideal solution would allow independent tuning of both the heuristic and the magnitude of search in all stages of the search tree.

3.2 Multistage Nested Rollouts

We present a multistage search technique utilizing sequences of nested rollouts. The sequence of heuristics is applied to the search tree in order. When the first heuristic reaches a local maximum, the second heuristic is applied, and so on until the last heuristic, which proceeds greedily until termination. A level of nesting, n, is associated with each heuristic, so that the coverage and search magnitude is tuned for each stage of the search tree. The multistage nested rollout algorithm is described in Figure 4(c).

Search proceeds in a similar manner to the nested rollouts algorithms. Children nodes are searched and evaluated with a recursive call to the search algorithm. When the value of a node s, is less than the backed up value of each node in the subtree generated from s (a local maximum is reached), the search at s is repeated with the next heuristic on the list. In its simplest form (n = 0 for all heuristics), a greedy path is chosen from the children nodes until reaching a local maximum, at which point the greedy search continues with the next heuristic. A graphical representation of this simple search mechanism with two heuristics can be seen in Figure 5(b).

Figure 7 displays a more complicated instantiation with two heuristics, h1 and h2. Search proceeds in a manner similar to that of the unstaged level-1 search (represented in Figure 6), with each child of the root searched with a level-0 search. When a local maximum is reached at a node s, a new search begins with the next heuristic. The choice made at the root will be based on the evaluations of all leafs reached by the final heuristic.

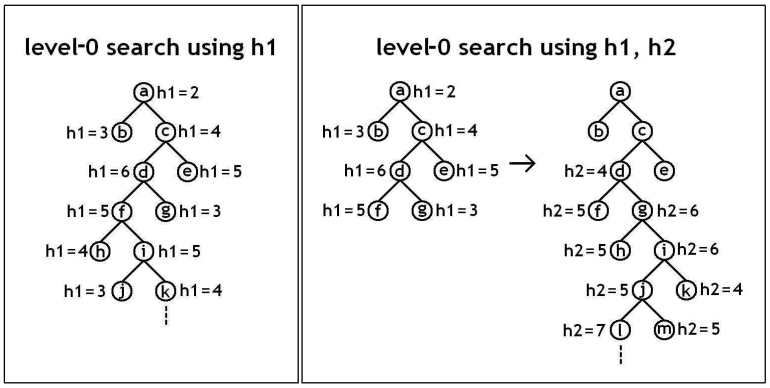

Figure 5: Representation of search paths taken for (a): level-0 nested rollouts (greedy search) and (b): multistage nested rollouts with two heuristics, each with level-0 search. When searching with multiple heuristics, the next heuristic is used when a local maximum is reached (e.g., node *d* in (b)). If no additional heuristics are available, search continues despite local maxima (e.g., node *d* in (a) or node *i* in (b)).

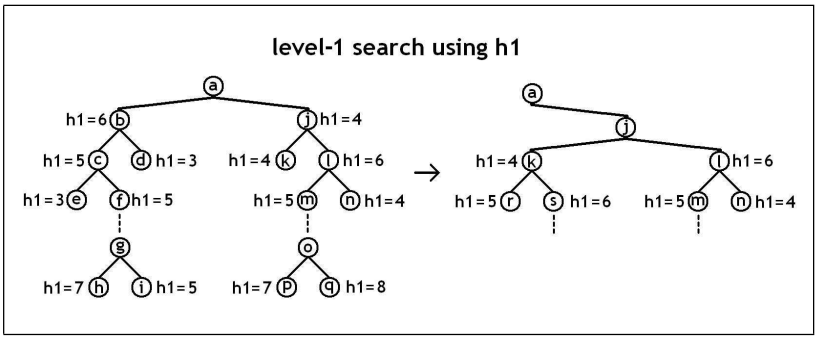

Figure 6: Representation of search paths taken for level-1 nested rollouts (standard rollouts). The results from level-0 search on the children of node *a* determine the eventual path.

4. THE THOUGHTFUL SOLITAIRE SOLVER

The multistage nested rollout algorithm searching through the compressed K^+ search tree described in Section 2 provides the basis for our real-time solver for Thoughtful Solitaire. The compressed tree reduces the depth of the search tree while increasing the branching factor. This increased branching factor allows the heuristics to evaluate a larger number of children at minimal cost. We have developed two heuristics (H1 and H2) for this problem, represented as weighted value functions in Table 1. (Please note the heuristics in this section are capitalized.)

The first heuristic, H1, is designed for opening the game, placing highest value on nearby search nodes lacking negative-valued features that complicate card movement (e.g., features 4, 5, 6). Cards located in the Foundation stacks (feature 1) receive a sliding-scale value, with the value decreasing as the **rank value** of the card increases, ensuring local maxima in H1's search. H2 is designed to search for simple paths to the goal state and places greatest value on nodes with face-up cards (feature 2) and cards in the Foundation stacks (feature 1).

Changes made to the basic multistage nested rollout algorithm make it especially effective in searching through our Solitaire domain. These changes include a straightforward pruning method, node caching, global and local loop prevention, and additional heuristics for action ordering.

4.1 Pruning

We prune branches of the search tree by searching a relaxed version of K^+ Solitaire, which can be done in time linear with the number of cards. Relaxed games without a solution will not have solutions in the standard game.

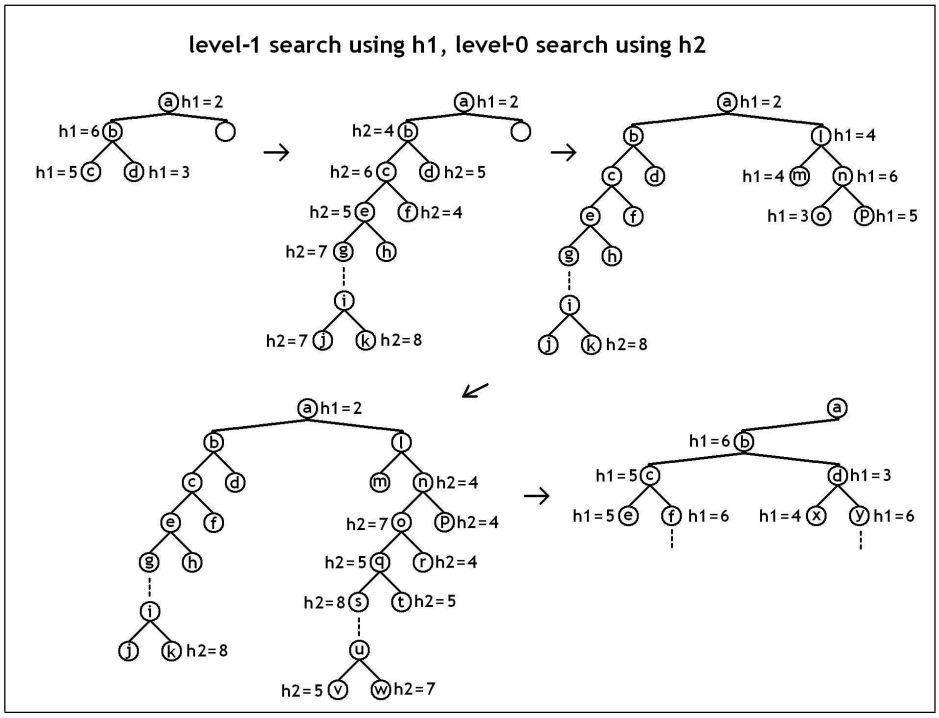

Figure 7: Representation of search paths taken for multistage nested rollouts with two heuristics: level-1 search with the first heuristic (h1), and level-0 search with the second (h2). Heuristics are switched when local maxima are reached (e.g., nodes *b* and *n*). The path to node *b* is chosen because *k* has a higher value than both *w* and *a*. If the leaf node values had been less than *a*, *a* would be a local maximum, and search with the second heuristic would begin at that node.

Num	Description of Feature for Card *x* (# of elements)	H1	H2
1	*x* is in a Foundation stack (52)	5 - **rank value**	5
2	*x* is face down in a Tableau stack (52)	**rank value** - 13	**rank value** - 13
3	*x* is available to be moved from the K^+ Talon (52)	0	1
4	*x* and the other card of same rank and colour are both face down in some Tableau Stack (26)	-5	-1
5	*x* is blocking a suited card of lesser rank (4×(12+11+...+1))	-5	-1
6	*x* is blocking one of its two Tableau *build cards* (48×2)	-10	-5

Table 1: The value functions used to determine the value of states in the search tree. H1 is designed for opening the game and H2 for end-game scenarios. The number of occurrences of each feature type is listed in parentheses.

The relaxed domain is formed by removing delete effects from actions. Once a card is available to be moved or played upon, it remains so throughout the entire search. This is a standard technique used to construct path-cost heuristics in general planning algorithms (Bonet, Loerincs, and Geffner, 1997; Hoffmann and Nebel, 2001). We do not include path costs into our heuristic, using it only for pruning purposes. A secondary dead-end pattern is also checked. This pattern involves states in which two cards of same colour and rank are in the same Tableau stack blocking both of their paths to the Foundation and one of their Tableau build cards. Two dead-end games are shown in Figure 8. The first, (a), is detected in the relaxed domain (notice the Q♠ blocking the K♡, K♢, and 2♠) while (b) is detected by the secondary pattern (the 10♡ and 10♢ are blocking the J♣, 9♢, and 5♡). Of 10 million sample games 855,741 were detected as dead ends using this algorithm. The empirical result increases the lower bound of dead-end games to 8.56% from a previous theoretical bound of 1.19% (Yan *et al.*, 2005).

(a) **(b)**

Figure 8: **(a)** A dead end detected in the relaxed domain. The Q♠ is blocking the K♡, K♢, and 2♠. **(b)** A dead end undetected in the relaxed domain, but detected as a special case pattern. After the 10♢ is built upon the J♠, the 10♢ and the 10♡ have no viable path to the Foundation stacks, as the 10♡ it is blocking the J♣, 9♢, and 5♡.

4.2 Caching Visited States

Each heuristic caches up to 5,000 visited nodes. The caches are filled with nodes that have been explored with a nested level greater than 0 (storing nodes at level 0 will quickly fill the cache with minimal savings). Cached nodes will be re-explored if visited with an increased degree of nesting.

4.3 Global Loop Prevention

The return path to the root node is checked against the current state. If the current node has been previously explored by the same heuristic at an identical nesting level, then the search is in an infinite loop. These loops are not guaranteed to be detected by caching alone because of the limited number of cached nodes.

4.4 Action Restrictions and Ordering

In order to speed search, some actions that duplicate other action sequences are prohibited from the search tree. There are three different types of actions that are prevented. In addition to this, the actions are ordered so that the most productive actions have priority.

Local Loop Prevention An action is eliminated from a state, *s*, in a search tree if, by definition, state *s* can be restored in a single action following that action. In order to maintain a complete search, these actions are included only when paired with a second action such that *s* cannot be trivially recovered.

Foundation Progression Cards in the Foundation are prohibited from moving back to the Tableau in the case that all cards of rank two less than its own are already in the Foundation (or if the card is an Ace or Two).

King to Tableau When multiple Tableau stacks are empty, only the furthest left is made open for a King (because they are all equivalent). When a King is already the bottom-most card in a Tableau stack, it is not allowed to move to an empty stack.

Action Ordering Actions are searched in the following order: (1) actions that move a card from Tableau to Foundation in which a face-down card is revealed, (2) actions that move a card to the Foundation, (3)

actions that move a card from Tableau to Tableau in which a face-down card is revealed, (4) actions that move a card from the Talon to the Tableau, (5) actions that move a card from the Foundation to the Tableau, and (6) actions that move a card from Tableau to Tableau without revealing a face-down card.

In addition to these modifications, a time-limit may be placed on the search algorithm. A time limit of 8 seconds is implemented in the downloadable game located at Bjarnason (2007). A more complete pseudo-code description of the actual solver is described in Figure 9.

```
procedure multistage-nested-rollout (s, h0, h1, ... hz, n0, n1 ... nz)
1    if s is the goal return WIN
2    if s is in a loop return LOSS
3    if time has expired or s is a dead-end or n0 is -1 return h0(s)
4    if this state has been cached for h0 and n0
5      if (z = 0) return h0(s)
6      else return multistage-nested-rollout(s, h1 ... hz, n1 ... nz )
7    while h0(s) is not LOSS
8      val = max_a (  multistage-nested-rollout (result(s,a), h0 ... hz, n0-1, ... nz)  )
9        and let a' be the maximizing action
10     if val is WIN return WIN
11     if val is LOSS or ( z≠0 and val < h0(s) )
12       if (z = 0) return h0(s)
13       else return multistage-nested-rollout (s, h1, ... hz, n1, ... nz)
14     s = result(s,a')
```

Figure 9: A description of the multistage nested rollouts algorithm, as implemented with search modifications specified in Section 4. On line 2, the history from the current state back to the root search node is checked for loops. On line 4, the current state is checked against the cache for the current heuristic and nesting magnitude.

5. EMPIRICAL RESULTS

We present performance results of the Thoughtful Solitaire Solver described in Section 4 with and without time constraints. All tests referenced in this article were performed on Dell Power Edge 1850 1U rack servers with Dual 3.4 Ghz Intel Xeon Processors with 2048 KB cache and 4GB of SDRAM.

5.1 Klondike Solitaire Lower Bounds

Using a multistage nested rollout search method with 3 levels of nested rollout search for H1 and a single rollout search (n=1) for H2, we were able to demonstrate that at least 82% of Thoughtful Solitaire (and therefore common Klondike Solitaire) games have winning solutions, with each game taking an average of 32 minutes to complete. Previous results by Yan *et al.* (2005), reported 70% wins on a version of Thoughtful Solitaire disallowing movement of partial Tableau stacks, taking 105 minutes per game. Table 2 compares the two methods at various levels of complexity. The search tree compression (see Section 2) significantly increases the branching factor of our search trees, and therefore increases the number of states from which higher levels of search are conducted. This largely explains the increases in both *Average Time Per Game* and *Win Rate* for single heuristic searches of our method (see Table 2(a)) compared to results reported by Yan *et al.* (2005) (see Table 2(b)).

5.2 Performance Under Time Constraints

Because individual games may take excessively long periods of time to complete their search, it is not practical to implement the unrestricted solver into a game intended for the casual user. It is necessary that a time limit be placed on the search of the tree. In order to demonstrate the feasibility of solving Thoughtful Solitaire (and Klondike Solitaire) in real time, a demo has been made available at Bjarnason (2007). In the demo, the search algorithm caps search times to 8 seconds. Table 3 presents time statistics of the unrestricted solver along side

Search Method	Win Rate (99% Conf.)	Games Played	Avg Time Per Game
n=0	16.17 ± 0.42	50,000	0.02 s
n=1	60.98 ± 0.56	50,000	0.74 s
n=2	74.03 ± 0.51	50,000	7.37 s
n_0=1;n_1=1	74.94 ± 0.50	50,000	3.64 s
n=3	77.99 ± 0.79	18,325	302.85 s
n_0=2;n_1=1	78.94 ± 0.47	49,956	212.38 s
n=4	79.17 ± 3.05	1,176	89.17 min
n_0=3;n_1=1	82.24 ± 2.71	1,323	31.67 min

(a)

Search Method	Win Rate (99% Conf.)	Games Played	Avg Time Per Game
Human	36.6 ± 2.78	2,000	20 min
n=0	13.05 ± 0.88	10,000	0.02 s
n=1	31.20 ± 1.20	10,000	0.67 s
n=2	47.60 ± 1.30	10,000	7.13 s
n=3	56.83 ± 1.30	10,000	96 s
n=4	60.51 ± 4.00	1,000	18.1 min
n=5	70.20 ± 8.34	200	105 min

(b)

Table 2: Results for rollout tests, from our solver (a), and as reported in Yan *et al.* (2005) (b) on a slightly stricter version of Solitaire. The degree of nesting is indicated by *n* when a single heuristic (H2) is used. Nesting degrees for multiple heuristics are indicated by n_0 for H1, and n_1 for H2. Results in (b) used a different heuristic which required a nested rollouts method slightly different from the method described in Figure 4. H1 and H2 are described in Table 1.

Search Method	Avg Time per Win	Avg Time per Loss	Avg Time Overall	% Wins $<$ 1 sec	% Wins $<$ 4 sec	% Wins $<$ 16 sec	% Wins $<$ 64 sec
n=0	0.06 sec	0.02 sec	0.02 sec	16.17	16.17	16.17	16.17
n=1	0.60 sec	0.95 sec	0.74 sec	49.03	60.38	60.97	60.98
n_0=1;n_1=1	1.35 sec	10.50 sec	3.64 sec	56.98	69.82	74.23	74.94
n=2	1.45 sec	24.29 sec	7.38 sec	52.38	67.88	73.23	74.03
n_0=2;n_1=1	0.22 min	16.00 min	3.54 min	56.87	70.47	76.11	78.06
n=3	0.29 min	21.89 min	5.05 min	53.41	69.22	75.33	77.14
n_0=3:n_1=1	0.44 min	176.27 min	31.67 min	57.82	72.26	78.31	80.57
n=4	2.12 min	420.02 min	89.18 min	55.44	69.98	74.92	77.21

Table 3: Results for tests under time constraints. The degree of nesting is indicated by *n* when a single heuristic (H2) is used. Nesting degrees for multiple heuristics are indicated by n_0 for H1, and n_1 for H2. Use of multiple heuristics wins a higher percentage of games in a shorter time period than using a single heuristic. H1 and H2 are described in Table 1.

percentage of games won for various instantiations of our solver, using either one or two heuristics. In Figure 10 we can see how the increased search in the first stage of search improves the performance over time, from 2^{-3} to 2^{10} seconds.

6. DISCUSSION

We have presented a novel real-time search algorithm designed for planning tasks with multiple stages. This multistage nested rollout algorithm allows the user to control the amount of search done at each level and specify a distinct heuristic for each level. The algorithm utilizes heuristic values from deep within the search tree to make action selections at every stage of search. Single stage rollout performance is guaranteed to perform at least as well as the heuristic policy (Bertsekas and Tsitsiklis, 1996), but this guarantee does not hold for our multistage nested rollout algorithm. Sequenced rollouts may direct the search down paths that a single heuristic would have never considered. We would like to develop a method that preserves this guarantee across multiple stages using multiple heuristics.

We have also presented a tree-compression method applicable to the games of Klondike Solitaire and Thoughtful Solitaire that eliminates the Stock, decreasing the depth of the search tree. An implementation of the multistage nested rollouts algorithm over this compressed search tree forms the foundation for a solver specific to the game of Thoughtful Solitaire. This solver has demonstrated that no less than 82% and no more than 91.44% of instances of Klondike Solitaire have winning solutions, leaving less than 10% of the games unresolved. We are confident that these bounds can be improved by using a more complex pruning method to decrease the upper bound and new heuristics to improve the lower bound. We would like to develop a method for learning optimal heuristics, through both feature generation and weight learning.

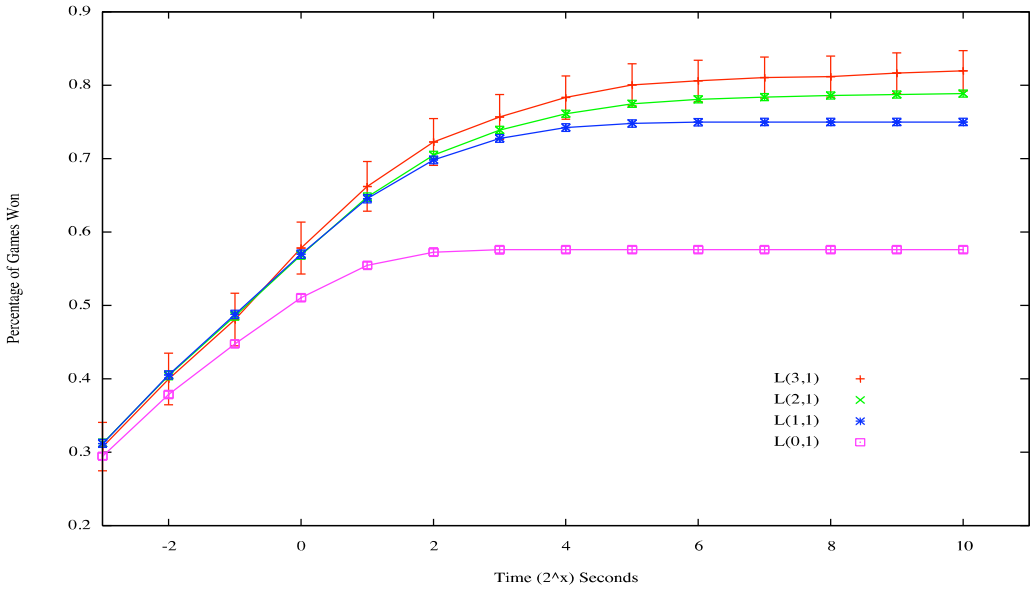

Figure 10: Comparison of winning percentage (as a function of time) for various levels of nesting of the multi-stage nested rollout algorithm. Error bars indicate 99% confidence intervals.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant IIS-0329278.

7. REFERENCES

Barto, A. G., Bradtke, S. J., and Singh, S. P. (1995). Learning to act using real-time dynamic programming. *Artificial Intelligence*, Vol. 72, Nos. 1–2, pp. 81–138. ISSN 0004–3702.

Baxter, J., Tridgell, A., and Weaver, L. (2000). Learning to Play Chess Using Temporal Differences. *Machine Learning*, Vol. 40, No. 3, pp. 243–263. ISSN 0885–6125.

Behnke, D. (2007). Deep Logic. http://www.codeplex.com/deeplogic.

Bertsekas, D. and Tsitsiklis, J. (1996). *Neuro-Dynamic Programming*. Athena Scientific, Boston, Massachusetts. ISBN 1–886529–10–8.

Bjarnason, R. (2007). K+ Solitaire. http://web.engr.oregonstate.edu/˜ronny/k.html.

Bonet, B. and Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, Vol. 129, Nos. 1–2, pp. 5–33. ISSN 0004–3702.

Bonet, B., Loerincs, G., and Geffner, H. (1997). A Robust and Fast Action Selection Mechanism for Planning. *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pp. 714–719, AAAI Press / MIT Press, Providence, Rhode Island. ISBN 0–262–51095–2.

Chang, H. S., Givan, R., and Chong, E. K. P. (2004). Parallel Rollout for Online Solution of Partially Observable Markov Decision Processes. *Discrete Event Dynamic Systems*, Vol. 14, No. 3, pp. 309–341. ISSN 0924–6703.

Cronin, D. (2007). Winnable Solitaire. http://winnablesolitaire.com.

Doug (2007). Cracking Solitaire. http://crackingsolitaire.blogspot.com.

Fish, S. (2005). Freecell Solver. http://www.shlomifish.org/freecell-solver/.

Hoffmann, J. and Nebel, B. (2001). The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, Vol. 14, pp. 253–302. ISBN 1558608575.

Korf, R. E. (1990). Real-time heuristic search. *Artificial Intelligence*, Vol. 42, Nos. 2–3, pp. 189–211. ISSN 0004–3702.

Kulow, S. (2006). More Info Revision 602069. http://www.commit-digest.org/issues/2006-11-12/moreinfo/602069/.

Tesauro, G. and Galperin, G. (1996). On-line Policy Improvement Using Monte-Carlo Search. *Advances in Neural Information Processing Systems 9*, pp. 1068–1074, MIT Press, Cambridge, MA. ISBN 0262100657.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, Vol. 38, No. 3, pp. 58–68. ISSN 0001–0782.

Yan, X., Diaconis, P., Rusmevichientong, P., and Van Roy, B. (2005). Solitaire: Man Versus Machine. *Advances in Neural Information Processing Systems 17* (eds. L. K. Saul, Y. Weiss, and L. Bottou), pp. 1553–1560, MIT Press, Cambridge, MA. ISBN 0262195348.

Yoon, S., Fern, A., and Givan, R. (2005). Learning Measures of Progress for Planning Domains. *Proceedings of the Twentieth National Conference of Artificial Intelligence (AAAI-05)*, pp. 1217–1222, AAAI Press, Menlo Park, CA. ISBN 157735236X.