Search in the Patience Game 'Black Hole'

Ian P. Gent [a], Chris Jefferson [b], Tom Kelsey [a], Inês Lynce [c], Ian Miguel [a], Peter Nightingale [a], Barbara M. Smith [d] and S. Armagan Tarim [d]

[a] *School of Computer Science, University of St Andrews, KY16 9SX, UK*
E-mail: {ipg,tom,ianm,pn}@dcs.st-and.ac.uk
[b] *Computing Laboratory, University of Oxford, UK*
E-mail:Chris.Jefferson@comlab.ox.ac.uk
[c] *IST/INESC-ID, Technical University of Lisbon, Portugal*
E-mail:ines@sat.inesc-id.pt
[d] *Cork Constraint Computation Centre, University College Cork, Ireland*
E-mail: {b.smith,at}@4c.ucc.ie

We present an evaluation of different AI search paradigms applied to a natural planning problem. The problem we investigate is a particular card game for one player called Black Hole. For paradigms such as SAT and Constraint Programming, the game has the particular advantage that all solutions are the same length. We show that a general version of Black Hole is NP-complete. Then we report on the application of a number of AI paradigms to the problem, namely Planning, Constraint Programming, SAT, Mixed-Integer Programming and a specialised solver. An important feature of Black Hole is the presence of symmetries which arise during the search process. We show that tackling these can improve search dramatically, as can caching states that occur during search. Our implementations as SAT, Constraint Programming and Planning problems are efficient and competitive, allowing detailed empirical evaluation of the strengths and weaknesses of each methodology. Our empirical evaluation shows that Black Hole is winnable approximately 87% of the time, and that given instances can be trivially solved, easy to solve, hard to solve and even intractable, depending on the AI methodology used to obtain solutions.

Keywords: Constraint Programming, Planning, Empirical Evaluation

1. Introduction

We propose patience games – card games for one player also known as "solitaire" – as a fruitful domain for studying search problems. These games are a natural Artificial Intelligence problem, since they are a recreation enjoyed and understood by many people, but for which computer-based solving techniques are generally not studied. There are hundreds of different patiences,[1] with many more variants derived by changing the number of piles or other features. The varied nature of these patiences will lead to different approaches being needed, and this study is designed to explore the range of validity of different AI techniques. In particular, as we show here, we can study a variety of mature AI paradigms applied to a single problem. A particular benefit to empirical analysis is that the problems of everyday concern are of a shuffled deck and are therefore entirely random. There is thus an effectively unlimited number of benchmarks available.

There is very little research on patiences: how to solve them, how to play them, and how winnable they are. The only body of work we know of is on the game Freecell. Exactly one of the 32,000 possible games in the original Windows program is unwinnable. Extensive empirical work has shown that the probability of the game being winnable is roughly 99.999%, and solvers are available for the game [4]. Freecell has also been used as a benchmark for AI planning programs [10]. General solvers could be very useful to players to detect insolubility or give hints. AI research can also feed back by helping the design of satisfying new patiences or variants of old ones, for example verify-

[1] The individual game called "Solitaire" in Windows™ is most properly called Klondike. There is a historical tendency for the name of the most popular patience (usually Klondike or Canfield) to become synonymous with the general pastime, and this can cause confusion.

AI Communications
ISSN 0921-7126, IOS Press.

ing that a proposed patience is solvable a reasonable percentage of the time.

The variety of patiences is likely to lead to a toolbox of techniques being required instead of a single one. For example, some games reveal all cards at the start and are open to analysis, while others enforce moves to be made with many cards remaining hidden. Other games are mixed, for example in an initial phase cards are placed before a final analytical stage where the cards placed in the first phase are played to a win (or not). Such different games are likely to be tackled in different ways.

In this paper we show the value of patience as a class of benchmark problems, using the game Black Hole as a case-study. We solve it using a variety of different AI paradigms, namely planning – described in Section 4, constraint programming (CP) – Section 5, propositional satisfiability (SAT) – Section 6, mixed integer programming (MIP) – Section 7, and a special-purpose solver – Section 8. We thus compare the advantages and disadvantages with respect to each other, while also being able to see the important features in each method. Black Hole is particularly appropriate for an initial study of AI techniques applied to patience: it is fully open, i.e. gives perfect information to the player at the start. Every successful game involves exactly 52 moves, making it easy to apply techniques such as constraint programming to it. As we show here, a natural generalisation of it is NP-complete and therefore we do not expect any shortcuts to be discovered which allow trivial solving in general. Finally, we report that on the problem of human interest, i.e. with 52 cards, Black Hole provides a challenge for all our methods.

We find some paradigms more effective than others in this paper, and give a detailed analysis of empirical results in Section 9. However, we absolutely do not claim that this shows the more successful techniques are better, even for Black Hole: our (relative) failures may simply be due to a lack of skill and ingenuity on our part. Instead, we intend our analysis to be a useful longitudinal study of a number of AI paradigms on a simple, but not trivial, problem of real interest to people. We emphasise the design decisions in each case and how they relate to the properties of the solvers used and the patience itself, proposing reasons for the techniques' success or failure.

2. Black Hole

Black Hole was invented by David Parlett [16] with these rules:

"*Layout* Put the Ace of spaces in the middle of the board as the base or 'black hole'. Deal all the other cards face up in seventeen fans [i.e. piles] of three, orbiting the black hole.

"*Object* To build the whole pack into a single suite based on the black hole.

"*Play* The exposed card of each fan is available for building. Build in ascending or descending sequence regardless of suit, going up or down ad lib and changing direction as often as necessary. Ranking is continuous between Ace and King."

The table below shows an instance of the game: the 18 columns represent the $A\spadesuit$ in the black hole and the 17 piles of 3 cards each.

	4♢	7♡	7♠	3♢	5♠	T♣	6♠	J♣
	9♠	9♡	J♡	4♠	K♢	Q♢	T♠	T♢
A♠	8♠	5♢	2♡	5♣	T♡	3♣	8♣	A♡

J♠	9♢	7♢	2♣	3♡	7♣	3♠	6♢	9♣
A♣	Q♠	K♠	Q♡	5♡	K♣	8♡	J♢	2♢
2♠	K♡	Q♣	4♡	6♣	6♡	A♢	4♣	8♢

A solution to this game is:

A♠2♣,3♠,4♢,5♠,6♠,7♠,8♡,9♠,8♠,9♣,T♠,J♠,
Q♡,J♡,T♣,J♣,Q♢,K♢,A♣,2♠,3♡,2♢,3♣,4♡,5♡,
6♣,7♡,8♣,7♣,6♢,7♢,8♢,9♡,T♡,9♢,T♢,J♢,Q♠,
K♠,A♡,K♡,Q♣,K♣,A♢,2♡,3♢,4♠,5♣,6♡,5♢,4♣.

We mention one general feature of search in Black Hole. The first two piles in the example layout both have 9s in the middle. If, at some point in the game, both the 4♢ and the 7♡ have been played, the two 9s are interchangeable *provided that* we don't need to play the 9♠ before the 9♡ to allow access to the 8♠, or the 9♡ before the 9♠ to access the 5♢. That is, the 9s are interchangeable if they are both played after both of their predecessors and before either of their successors. In these circumstances, we can choose the order in which the two 9s are played and not backtrack on this choice. Such a symmetry, dependent on what happens in search, is called an *almost symmetry* in AI planning [6], or a *conditional* symmetry in constraint programming [8,7]. How to deal with this plays an important role in a number of sections to follow.

Our experimental evidence will show that Black Hole has a 87.5% chance of being winnable with

perfect play. About 2.9% of games are trivially unsolvable, because no deuces or Kings are available in the top layer[2], but of course there are non-trivial ways to be unsolvable.

3. NP-Completeness of Generalised Black Hole

To show that Black Hole is NP-complete, we consider a generalised version where there are any number of card values, fans can have arbitrary and differing sizes, and each card value can have a different number of suits. We show instances of the SAT problem can be encoded using this generalised version of Black Hole.

Consider a SAT instance with list of variables V and list of clauses C where $|C| = n$. Then the deck to be used is constructed as follows. Note throughout this section the particular suit of a card is only given when relevant, and that subscript i indicates the i-th clause.

1. For each $v \in V$, there are two cards of equal value and different suit, denoted vT and vF, called literal cards.
2. For each clause there are (clause size + 1) different suits for the card values c_i and g_i. The c_i are called clause cards, the g_i are called gate cards. We need (clause size + 1) different suits to construct gate fans for each clause, as described later.
3. Cards with values o_1, o_2, o_3 and sf where o_1, o_2 and o_3 have two suits, and sf has three. o_1, o_2, o_3 take values distinct from the c_i, and are used to construct a 'one-way trap' through which a solution may pass in one direction only. sf denotes a reserved start/finish card value.

A card can only be played onto the black hole if its value is adjacent to the card on top of the black hole. Given an ordering $\mathcal{V}$ on the variables, the ordering on the card values begins with the start/finish card sf, followed by the cards representing literals honouring the order $\mathcal{V}$, followed by the c_i and g_i cards in order $c_1, g_1, c_2, g_2, \ldots, c_n, g_n$, followed by the cards o_1, o_2, o_3 which are used to construct the one-way trap. Hence, for example, if there are three clauses and three variables (a, b & c, say), and we choose sf to be an Ace, then

- aT and aF are deuces;
- bT and bF are threes;
- cT and cF are fours;
- $c_1, g_1, c_2, g_2, c_3, g_3$ take values 5, 6, 7, 8, 9 and 10 respectively;
- o_1, o_2, o_3 are a Jack, a Queen and a King respectively.

Setting up the cards

The beginning position is constructed as follows:

- **Literal fans:** For each literal card L, a fan is constructed with L on top, and a copy of c_i for each clause that contains L. The c_i cards are ordered with smallest i nearest the top of the fan.
- **Gate fans:** For each clause card c_i, there is a fan which has a copy of g_i at the bottom, then a copy of c_i, then a g_i for each occurrence of c_i in the literal fans.
- **One way trap fan:** A fan consisting of

$$o_1, o_2, o_3, sf, o_1, o_2, o_3, sf.$$

The main idea behind the encoding is that exactly one of vT or vF will be chosen for each SAT variable v. This will reveal a number of clause cards in the literal fans. If some clause is not represented in the uncovered literal fans, then no path can exist to the o_1 card. Otherwise, a path such as $c_1, g_1, c_1, g_1, c_2, g_2, \ldots, c_n, g_n, o_1$ exists. Notice that the gate cards allow more than one of each clause card to be collected.

Following this, o_1, o_2, o_3 are picked up for the first time and form a one-way trap. If this point can be reached, it is always possible to have reached it in such a way that it will now be possible to pick up all remaining cards and then go through o_1, o_2, o_3 once again and finish.

As an example of deck construction, consider the formula $(a \vee b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c)$, whose mapping into Black Hole is given below.

aT	aF	bT	bF	cT	cF
c_1	c_2	c_1	c_3		c_1
c_3		c_2			c_2
					c_3

[2] For this to happen, the 17 top cards in each fan must be chosen from 43, i.e. 51 without the 2's or K's. $\binom{43}{17}/\binom{51}{17} = 0.0285\ldots$.

g_1	g_2	g_3	o_1
g_1	g_2	g_3	o_2
g_1	g_2	g_3	o_3
c_1	c_2	c_3	sf
g_1	g_2	g_3	o_1
			o_2
			o_3
			sf

The initial card in the black hole is sf. The first part of a solution sequence for this instance (corresponding to $a = True, b = True, c = True$) is sf, aT, bT, cT. At this point a number of clause cards are now visible. The solution now picks up as many clause cards as possible from the uncovered literal fans. The fans of gate cards are used to acquire as many occurrences of each clause card as possible, but none of the clause cards from the gate fans are taken. Finally o_1, o_2, o_3, sf is picked up for the first time:

$$c_1, g_1, c_1, g_1, c_2, g_2, c_3, g_3, c_3, g_3, o_1, o_2, o_3, sf.$$

Since the sequence has passed sf, the sequence is allowed to begin its second part, which clears up all the remaining cards. At this point the clause cards in the gate fans are collected:

$$sf, aF, bF, cF, c_1, g_1, c_1, g_1, c_2, g_2, c_2, g_2, c_2, g_2, c_3, g_3, c_3, g_3.$$

Finally o_1, o_2, o_3, sf are collected to complete the sequence.

3.1. Proof of correctness

Definition 1 The language of instances of Black Hole is defined as follows. In the following, each card is specified as a rank and a suit. A problem instance is a sequence, beginning with the card initially in the black hole, followed by a punctuation symbol, followed by the fans in arbitrary order, separated by the punctuation symbol. A witness is simply a single sequence containing all cards once, starting with the card initially in the black hole.

Definition 2 The initial part of a solution sequence of a Black Hole encoding consists of any cards which come before the first occurrence of the sequence o_1, o_2, o_3.

Lemma 1 The initial part of the solution sequence of a Black Hole encoding must contain exactly one of the cards vT and vF, for each variable v.

The deck contains two cards for each SAT variable v, vT and vF. The solution starts at the card sf and must contain the sequence o_1, o_2, o_3 twice. Therefore the complete sequence must look like this: $sf, X, o_1, o_2, o_3, sf, Y, o_1, o_2, o_3, sf$ where X and Y are subsequences which contain at least one copy of each of the literal cards. Since the sequence must pass through all the literal cards twice, there must be only one literal card of each value in X.

Lemma 2 If the encoded SAT instance is satisfiable, there is a route from the initial card to the first occurrence of o_1.

By Lemma 1, such a sequence must contain exactly one occurrence of each literal card value. If we take the literal cards which represent a solution to the encoded SAT instance, then the fans they were on will contain at least one occurrence of each clause card c_i. There are sufficient available g_i cards in the gate fans to form pairs with all these c_i cards, and the c_i cards are sorted in ascending order in the fans. Hence there is a route where each c_i, g_i pair occurs.

Lemma 3 If the encoded SAT instance is satisfiable, then there is a solution to the Black Hole encoding

By Lemma 2, it is possible to get from the beginning to the first occurrence of o_1. Note also during this sequence it was not necessary to take any clause cards from the gate fans, and we can therefore assume that no such cards were taken. The remainder of the solution begins o_1, o_2, o_3, sf then proceeds through all the remaining literal cards, then all the remaining c_i, g_i pairs in ascending order, where there is at least one of each left in the gate fans.

Lemma 4 If there is a solution to Black Hole instance generated from a SAT instance, the SAT instance is satisfiable.

If there is a solution to the Black Hole instance then by Lemma 1, the initial part of this solution must contain exactly one of the cards vT or vF for

each variable v. These literals form a solution to the encoded SAT instance as the fans with these cards must between them contain at least one occurrence of c_i for every clause i, and therefore each clause in the SAT instance must be satisfied by this assignment.

Lemma 5 Black Hole is NP-Easy.

For the Black Hole language described in Definition 1, it is clear that the solution size is linear in the size of the problem instance, because they contain the same number of cards. A solution can be checked in polynomial time by playing the game: play cards into the black hole in the order specified by the witness. If, at any stage, some card is not available to be played, the witness is invalid, otherwise it is valid.

Theorem 1 Black Hole is NP-complete.

The combination of Lemmas 3 and 4 show that a SAT instance is satisfiable if and only if its Black Hole encoding is satisfiable, hence the encoding is correct. The encoding produces a Black Hole instance of a size that is linear in the number of literals in the SAT formula, therefore Black Hole is NP-hard. Combined with Lemma 5, we prove that Black Hole is NP-complete.

The language used to describe a Black Hole instance may not be the most compact. If another language which is logarithmically more compact were found, the witness would be exponentially larger than the instance, and Black Hole would not be NP-easy. However, since we polynomially reduce SAT into Black Hole, such a result would also prove that SAT is not NP-easy.

While this proof has considered encoding a general SAT instance, it could of course encode specialisations of SAT, in particular 3-SAT with a maximum of 5 occurrences of each variable, which is itself NP-complete. This is interesting because encoding this problem would put a fixed limit on the maximum size of the fans (to 8) and the number of suits (to 4). Ranks and number of fans remain unlimited.

4. An AI Planning Encoding

In AI planning, an initial state is gradually transformed into a goal state through the application of plan operators [1]. Black Hole can straightforwardly be characterised in this way: the stacks and initial hole card comprise the initial state, the goal state is that all cards are played, and a move is to play a card. Hence, it is natural to test the performance of AI planning systems on this domain.

The plan objects in our PDDL [5] encoding are simply the ranks (ace–king) and the suits (spades, clubs, diamonds, hearts). Each card is specified by a combination of rank and suit objects. We take this approach, as opposed to a single object per card, to simplify the description of adjacency. The initial, current and goal states are described using a number of simple propositions. (`hole` *rank*) indicates the rank of the card currently in the hole — note that it is not necessary to know the suit of the hole card. (`unplayed` *rank suit*) and (`played` *rank suit*) are self-explanatory. We use both because some planners do not accept negated preconditions and/or goals. (`top` *rank suit*) indicates that a particular card is at the top of a stack in the *initial* state. Similarly, (`under` $rank_1$ $suit_1$ $rank_2$ $suit_2$) indicates that, in the initial state, the card denoted by $rank_1$ $suit_1$ is underneath that denoted by $rank_2$ $suit_2$. We make use of two further variations of `under`, `underSameSuit` and `underSameRank`, since most AI planners forbid more than one parameter associated with an operator from being instantiated to the same plan object. Finally, (`plusone` $rank_1$ $rank_2$) indicates that $rank_1$ is adjacent below $rank_2$. Thirteen such propositions describe the adjacency of the set of ranks.

The decision to use `plusone` necessitates both Play-Up and Play-down operators. The simplest case is in playing a card that is on top of a stack initially, since there is no need to check that the card above has been played. For illustration, Play-Up-Top is given below. Space precludes presenting the full set of operators,

```
(:action PLAY-UP-TOP
 :parameters (?rank ?suit ?hole)
 :precondition (and (top ?rank ?suit)
                    (hole ?hole)
                    (plusone ?hole ?rank)
                    (unplayed ?rank ?suit))
 :effect (and (not(unplayed ?rank ?suit))
              (played ?rank ?suit)
              (not(hole ?hole))
              (hole ?rank)))
```

The other operators follow the same basic pattern. There are 12 operators in all (6 in either direction).

Five of each six deal with playing a card that was not on the top of a stack initially, with the need to avoid instantiating different parameters with the same plan object accounting for the variations.

Naively, the goal can be specified as all cards having been played. This is needlessly complex. It suffices to say that the bottom card of each stack must be played, since this implies that all cards above must also have been played. This reduces the number of goal conditions from 51 to 17 in the standard game.

We experimented with two state-of-the-art AI planning systems, Blackbox 4.2 [14] and FF 2.3 [12]. Blackbox is a Graphplan-based [3] planner that transforms the planning graph into a large propositional satisfiability problem. The solution to this problem, which is equivalent to a valid plan, is obtained by using the CHAFF [15] dedicated SAT solver. FF is a forward-chaining heuristic state-space planner that generates heuristics by relaxing the problem and solves using a Graphplan-style algorithm. Preliminary experimentation revealed that, on this encoding, FF performed by far the better. Hence, we focused on the use of FF.

4.1. Computational experience

We ran the FF planner on 2,500 randomly chosen problems. Over 80% were solved in under one second, although about 2% of the problems timed out (no result after 2844CPU seconds). Interestingly, the maximum non-timed out solution time was 130 CPU seconds, so it appears that there is a small percentage of problems for which our planning approach is not suited. This observation is reinforced by the lack of correlation between the problems that FF found hardest to solve and the problems that our SAT and Constraint Programming solvers found hardest. A more detailed comparison of results is given in Section 9.

5. A Constraint Programming Model

Constraint Programming (CP) is a powerful method for solving difficult combinatorial problems. Problems are characterised by a set of decision variables and a set of constraints that a solution must satisfy, and are then solved by search. We can represent a solution to the game as a sequence of the 52 cards in the pack, starting with the ace of spades, the sequence representing the order in which the cards will be played into the Black Hole. This makes it easy to devise a basic CP model. In fact, it is a permutation problem [11]: if the cards are numbered 0 (the ace of spades) to 51, the sequence of cards can be represented as a permutation of these numbers. So we can have two sets of dual variables: x_i represents the ith position in the sequence, and its value represents a card; y_j represents a card and its value is the position in the sequence where that card occurs. We have the usual channelling constraints: $x_i = j$ iff $y_j = i$, $0 \leq i, j \leq 51$. We set $x_0 = 0$.

The constraints that a card cannot be played before the card above it, if there is one, has been played are represented by $<$ constraints on the corresponding y_j variables. The constraints that each card must be followed by a card whose value is one higher or one lower are represented by constraints between x_i and x_{i+1} for $0 \leq i < 51$. We use a table constraint for this, i.e. the constraint is specified by a list of allowed pairs of values.

The variables $x_0, x_1, ..., x_{51}$ are the search variables: the variables $y_0, y_1, ..., y_{51}$ get assigned by the channelling constraints. The x_i variables are assigned in lexicographic order, i.e. the sequence of cards is built up consecutively from start to finish. There is scope for value ordering, however (see below). This simple model using only binary constraints models the problem successfully, but in practice search is prohibitive. We have therefore correctly investigated other techniques which make search practical.

We first deal with the conditional symmetry [7] described in Section 2. Recall that in the example the 9♠ and the 9♡ are interchangeable if both have been played after the cards above them, the 4♢ and 7♡, and before the cards immediately below them, 8♠ and 5♢. To break this conditional symmetry, we can add the constraint: if 4♢ $<$ 9♡ and 9♠ $<$ 5♢ then 9♠ $<$ 9♡. This constraint forces 9♠ to be played before 9♡ when they are interchangeable. Given any ordering of the occurrences of each value, all constraints of this form can be added, pairwise, before search. This does not change the solutions returned if (as we describe below) the same order of occurrences is preferred by the value ordering heuristic. The constraints are simplified if the preferred card of the pair is at the top of its pile or the other card is at the bottom of its pile, or both. In particular, if the preferred card is at

the top of its pile *and* the other card of the pair is at the bottom of its pile, then we can add a simple precedence constraint that the preferred card must be placed before the other. Because the conditional symmetry breaking constraints are designed to respect the value ordering, the solution found is the same as the solution that would be found without the constraints. The constraints simply prevent the search from exploring subtrees that contain no solution. Hence, the number of backtracks with the constraints is guaranteed to be no more than without them. Furthermore, they appear to add little overhead in terms of runtime; they cannot become active until their condition becomes false on backtracking, and they then become simple precedence constraints that are cheap to propagate. It is difficult to give statistics to show the difference that conditional symmetry breaking constraints makes to the performance of the solver: we have been able to solve few random instances without them, given a run-time limit of 10 minutes per instance. For those that can be solved within 10 min, adding condition symmetry breaking constraints can make orders of magnitude difference to the search effort and runtime. For example, a particular instance took 336,321 backtracks and 326 sec. to solve without them; when they were added, this was reduced to 252 backtracks and 0.66 sec.

Initially, when a variable x_i is selected for assignment, we simply selected its values in an arbitrary order (spades first in rank order, then hearts and so on.) We then changed the value ordering, so that cards in the top or middle layers are chosen before cards of the same value lower down in the initial piles. This fits with the problem, in that it makes sense to clear off the top layer of cards as quickly as possible. This also is consistent with the conditional symmetry breaking constraints: as long as values in the same layer are considered in the same order by the heuristic as in the constraints, the same solutions will be returned first with or without the conditional symmetry breaking constraints. Since this is a heuristic, it is not guaranteed to reduce search on each individual instance, but overall, it does reduce search considerably, by about an order of magnitude.

The above CP model has been implemented in ILOG Solver 6.0 and applied to 2,500 randomly generated instances. The performance of the CP model is highly skewed: half of the instances take fewer than 100 backtracks to solve, or to prove unsatisfiable, whereas the most difficult instances take millions of backtracks. This is shown in Figure 1, where the instances are sorted by search effort.

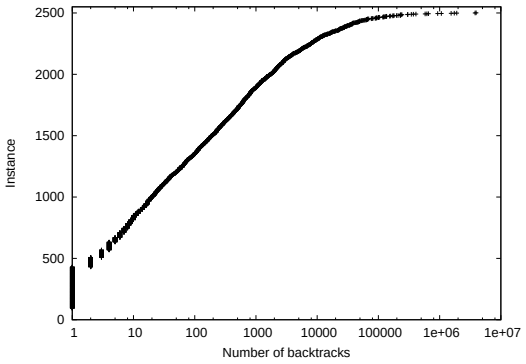

Fig. 1. Number of backtracks to solve 2,500 random instances of 'Black Hole'.

5.1. Caching states in 'Black Hole'

We now show that it can be worthwhile to cache information about the assignments visited during the search for solutions: this information can be used to prune parts of the search visited later and avoid wasted effort, as described in [18].

When a constraint satisfaction problem (CSP) is solved by depth-first backtracking search, and the search backtracks, the failure of the current assignment is due to some inconsistency that is not explicitly stated in the constraints. The search has discovered that the assignment cannot be extended to a solution; it is a *nogood*. There is no point in recording the assignment itself, in order to avoid it in future, because the search will never revisit it anyway. However, in some problems, assignments can occur later in the search that are *equivalent* to the failed assignment, in the sense that they leave the remaining search in the same state, and hence whether or not the equivalent assignment will fail can be determined from the failed assignment.

In such a case, if assignments are recorded and an assignment occurs later in the search that is equivalent to one that has already failed, the search can immediately backtrack without rediscovering the same failure. Such equivalent states occur in our CP model for Black Hole. At any

point during search where the current assignment is about to be extended, a valid sequence of cards has been built up, starting from the ace of spades. Whether or not the sequence can be completed depends only on the cards that have been played and the last card; apart from the last card, the order of the previously-played cards is immaterial.

For instance, suppose the following sequence of cards occurs during search (assuming that in some game the sequence is possible, given the initial layout of the cards):
A♠-2♣-3♠-4♢-5♠-4♣-3♣-2♠-A♣-K♢-A♢-2♢-3♢
If at some later point in the search, the following sequence occurs:
A♠-K♢-A♢-2♣-3♠-2♠-A♣-2♢-3♣-4♣-5♠-4♢-3♢
the second sequence will not lead to a solution. The set of cards in both sequences is the same, and they end with the same card. Hence, in both cases, the remaining cards and their layout are the same. Since the first sequence did not lead to a solution (otherwise the search would have terminated), the second will not either.

Based on this insight, the search algorithm in Solver has been modified to record and use the relevant information. The search seeks to extend the current sequence of cards at *choice points*. Suppose that the first unassigned variable is x_k and the values of the earlier variables are $x_0 = 0, x_1 = v_1, ..., x_{k-1} = v_{k-1}$. (Some of these values may have been assigned by constraint propagation rather than previous choices.) The search is about to extend this assignment by assigning the value v_k to x_k. A binary choice is created between $x_k = v_k$ and $x_k \neq v_k$, for some value v_k in the domain of x_k. The set of cards played so far, $\{v_1, v_2, ..., v_{k-1}\}$ and the card about to be played, v_k, are then compared against the states already cached. If the search has previously assigned $\{v_1, v_2, ..., v_{k-1}\}$ to the variables $x_1, x_2, ..., x_{k-1}$, in some order, and v_k to x_k, then the branch $x_k = v_k$ should fail. If no match is found, a new state is added to the cache, consisting of the set of cards already played and the card about to be played, and the search continues. In the example, when the 3♢is about to be added to the sequence, the set {2♠, 3♠, 5♠, A♢, 2♢, 4♢, K♢, A♣, 2♣, 3♣, 4♣}, and $x_{12} = 3♢$, would be compared with the states already visited.

The implementation represents the set of cards in the current sequence, excluding the A♠, as a 51-bit integer, where bit $i = 1$ if card i is in the set, $1 \leq i \leq 51$. The current state can only match a state in the cache if both the number of cards played $(k-1)$ and the current card (v_k) match. Hence, the cache is indexed by these items. It is stored as an array of extensible arrays, one for each possible combination of $k-1$ and v_k: this is a somewhat crude storage system, but has proved adequate for this problem. Within the relevant extensible array, the integer representing $\{v_1, v_2, ..., v_{k-1}\}$ is compared with the corresponding stored integers, until either a match is found, or there is no match. In the former case, the search backtracks: the current state cannot lead to a solution. Otherwise, the integer representing $\{v_1, v_2, ..., v_{k-1}\}$ is added to the array, $x_k = v_k$ is added to the sequence being built and the search continues.

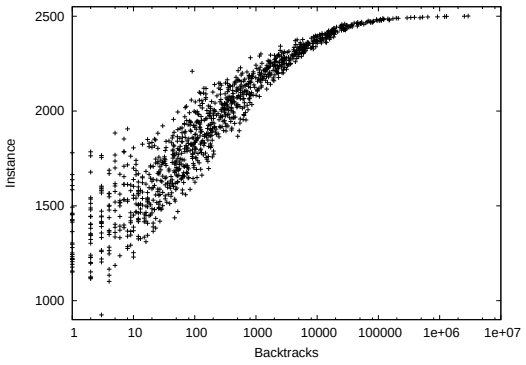

Fig. 2. Solving 2,500 random instances of 'Black Hole': difference in number of backtracks between the original search and the search with cached states, instances in the same order as Figure 1.

Figure 2 shows the reduction in the number of backtracks required to solve the 2,500 instances resulting from caching states. Only the instances which take fewer backtracks with caching than without are shown, but the instances are given the same numbering as in Figure 1 (so that the most difficult instance from Figure 1 is still shown as instance 2,500). It is clear that the saving in search effort increases with the search effort originally expended.

For all but 15 of the 1,206 instances that take 50 or fewer backtracks to find a solution, caching states visited makes no difference to the search effort. However, since few states are cached in these cases, the run-time is hardly affected either. ILOG

Solver occasionally reports a longer run-time with caching than without, by up to 0.01 sec., but only for instances that take little time to solve in either case.

At the other end of the scale, the instances that take more than 1 million backtracks with the original search are shown in Table 4; these instances have no solution. For these instances,

Table 1

Number of backtracks and run-time in seconds (on a 1.7GHz Pentium M PC, running Windows 2000) to solve the most difficult of the 2,500 'Black Hole' instances, with and without caching states visited.

No caching		Caching	
Backtracks	Time	Backtracks	Time
3,943,901	1,427.93	1,020,371	431.33
3,790,412	1,454.16	1,259,151	509.94
1,901,738	721.07	606,231	251.01
1,735,849	681.57	528,379	233.40
1,540,321	582.71	619,735	257.95
1,065,596	398.44	423,416	176.01

caching states visited reduces the search effort by at least 60%; for the most difficult instance, the reduction is nearly 75%. In spite of the unsophisticated storage of the cache, the saving in run-time is nearly as great; more than 55% for all six instances, and 70% for the most difficult instance.

To show more clearly how caching affects the search, Figure 3 shows the search profile for the most difficult instance of the 2,500 for the original search. The number of choice points is plotted against the number of variables assigned when the choice point is created, so showing the depth in the search where the choice point occurs. The number of cached states at each depth is also shown; this is equal to the number of choice points where no matching state is found in the cache and the search is allowed to continue.

When the search backtracks because the current state matches one already visited, a whole subtree that would have otherwise been explored is pruned. This is why the reduction in choice points as a result of caching, shown in Figure 3 is much greater than the number of choice points that match cached states; it also explains why, without caching, choice points tend to occur deeper in the search.

The total number of cached states for the instance shown in Figure 3 is about 1.25 million

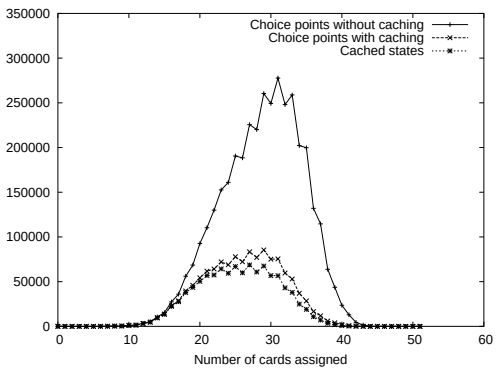

Fig. 3. Proving insolubility for the most difficult 'Black Hole' instance in the sample, with and without caching.

($< 2^{21}$). In a permutation problem, the number of possible assignments is at most the number of subsets of the values, i.e. 2^n, where n is the length of the sequence, in this case effectively 51; hence, this is an upper bound on the number of states that need to be cached during the course of search. However, in this case, most of the subsets of the cards are not feasible states, since a valid sequence cannot be constructed in which the cards follow each other correctly in ascending or descending rank. Hence, the number of possible cached states is much less than 2^{51}, even for the difficult unsatisfiable instances.

6. A SAT Model for Black Hole Patience

Propositional Satisfiability (SAT) is a technique closely related to CP in which the domains of all variables are Boolean and the constraints are expressed in conjunctive normal form. Specialised solvers achieve large efficiency gains by exploiting the simplicity of this specification language.

The SAT model is conceptually similar to the CP model, although additional variables are needed to achieve the same expressiveness.

First of all, we have a 52×52 matrix M of variables, where M_{ij} is true if card i is played into the black hole in the j^{th} position. We know in advance that the ace of spades is in the first position. The constraint that each card is played exactly in one position is represented by at-least-one and at-most-one clauses. Also, clauses are placed on the

variables in M to ensure that each card is followed by a card whose rank is one value higher or lower. When a solution is found, these variables are used to obtain the solution.

In addition, a second matrix with the same size establishes the order relations between cards. For establishing the order relations, we use a *ladder* matrix [9,2] i.e. a matrix in which for each row we must have a sequence of zero or more true assignments, and all following variables are assigned false. The first entry to have value false gives the position where the respective card has been played. Observe that the entries in this matrix are easily related with the entries in the first matrix. Besides the clauses to guarantee that only valid assignments are allowed, on this matrix clauses are added to guarantee that a card is played into the black hole only after all the cards above it have been played.

Finally, a third *ladder* matrix is required for applying symmetry breaking, where in this matrix the columns contain a sequence of zero or more true assignments, followed by all variables being assigned false. Clauses are placed on this matrix to eliminate conditional symmetries, i.e. search states where cards of the same rank are interchangeable. The conditions under which these symmetries arise have already been described for the CP model.

Unlike CP solvers, most SAT solvers do not provide the option of specifying a variable or value ordering and therefore this part of the CP encoding does not transfer to SAT.

Experimental results – described in Table 2 and obtained using the siege SAT solver [17] – demonstrate that SAT is indeed a competitive approach for solving the black hole problem. Although the CP solver is usually faster, the SAT solver is definitely more robust. We discuss the relative merits of approaches more fully in Section 9.

Instance	Mean SAT nodes	Mean CPU time
01	33,429	11.859
09	2,876	0.156
19	1,902	0.071
238	5,080	0.526
337	44,431	22.670
635	59,744	33.075
642	11,293	1.701
1360	1,595	0.028
1698	3,465	0.240
2223	16,721	4.188

Table 2
Sample SAT solver performance – siege with 50 seeds

7. Mixed Integer Programming Models

Integer programming (IP) is a powerful tool for solving linear optimization problems with discrete decision variables. It has been applied successfully to a variety of fixed-length AI planning problems (e.g. [13]). In this section, we present four different mixed integer programming models (MIP1-4) for Black Hole and investigate their computational performances on a test suite.

In representing an instance with 17 stacks of 3 cards each, the top cards are numbered 1 to 17, the middle cards from 18 to 34, and the bottom cards from 35 to 51. Notation is as follows: • $a[i,j]$ denotes a binary decision variable matrix of size 51×51, in which the cell (i,j) is 1 if the card in the ith position is removed during the jth move, otherwise 0. • $b[i,j]$ is a binary decision variable matrix of size 51×4, in which the cell (i,j) denotes the difference between the values of two consecutive cards played at moves i $(i = 0, ..., 50)$ and $i+1$. In this notation j denotes "the type of difference", which could be only one of these values: 1 $(j = 1)$, -1 $(j = 2)$, 12 $(j = 3)$, and -12 $(j = 4)$. • $v[i]$, a parameter array whose ith element denotes the value of the card in the ith position. • $x[i]$ is a continuous decision variable (but takes only integer values) and denotes "the move number" for the card at position i (i=1,...,51). • $y[i]$ is a continuous decision variable (takes only integer values) denoting the value of the card ($y[i] \in \{1, ..., 13\}$) played at the ith move. $y[0]$ is 1.

In what follows we present four different MIP formulations of the "Black Hole" game and compare the computational performance of these models.

7.1. MIP-1

The first MIP model (MIP-1) consists of two sets of constraints: a card may only be played after those on top of it have been played, and successively played cards must differ in value by either 1 or 12.

$$x[i+17(j-1)]+1 \leq x[i+17j],\ i=1,...,17; j=1,...,2 \quad (1)$$

$$x[i]+2 \leq x[i+34],\ i=1,...,17 \quad (2)$$

$$x[i] = \sum_{k=1,...,51} k.a[i,k],\ i=1,...,51 \quad (3)$$

$$\sum_{k=1,...,51} a[i,k] = 1,\ i=1,...,51 \quad (4)$$

$$\sum_{i=1,...,51} a[i,k] = 1,\ k=1,...,51 \quad (5)$$

$$y[i] = \sum_{j=1,...,51} v[j].a[j,i],\ i=1,...,51;\quad y[0]=1 \quad (6)$$

$$y[i]-y[i+1] = \\ -b[i,1]+b[i,2]-12b[i,3]+12b[i,4],\ i=0,...,50 \quad (7)$$

$$\sum_{j=1,...,4} b[i,j] = 1,\ i=0,...,50 \quad (8)$$

$$1 \leq y[i] \leq 13,\ 1 \leq x[i] \leq 51,\ i=1,...,51 \quad (9)$$

In the above formulation, Eqs.(1) and (2) express that, in any stack, the top card must have been played before an underneath card can be played. Eq.(3) sets the relation between $x[i]$ and $a[i,j]$. Since $a[i,j]=1$ denotes that the card i is played at move j, then $x[i]$, which is the ith card's move number, must be equal to $j.a[i,j]$. Clearly, each card can be played only once (Eq.(4)) and there should be one card assigned to each move (Eq.(5)). Eq.(6) has a similar function to Eq.(3) and sets the relation between $y[i]$ and $a[j,i]$: for card j which is played at move i we have $a[j,i]=1$ and must have $y[i]=v[j]$. Eq.(7) expresses the rule that the difference between the values of consecutively played cards must be an element of the set $\{1,-1,12,-12\}$. It is clear that only one of these four values can be the difference (Eq.(8)). Eq.(9) sets the lower and upper bounds for the continuous decision variables. From Eq.(3) and Eq.(6) we see that the right hand side of the equalities may only assume integer values and therefore, although $x[i]$ and $y[i]$ are declared as continuous decision variables, they never take on non-integer values.

The above formulation can be improved by replacing Eq.(7) with a set of constraints providing tighter relaxation. A set of such constraints are given below:

$$y[i+1]-y[i] \leq -11b[i,1]+12 \quad (10)$$

$$y[i+1]-y[i] \geq 13b[i,1]-12 \quad (11)$$

$$y[i]-y[i+1] \leq -11b[i,2]+12 \quad (12)$$

$$y[i]-y[i+1] \geq 13b[i,2]-12 \quad (13)$$

$$12b[i,3]+1 \leq y[i+1] \quad (14)$$

$$12(1-b[i,3])+1 \geq y[i] \quad (15)$$

$$12b[i,4]+1 \leq y[i] \quad (16)$$

$$12(1-b[i,4])+1 \geq y[i+1] \quad (17)$$

Eqs.(14)–(17) give us a very strong formulation compared to Eq.(7). Let us assume that $b[i,3]=1$ (the difference between the card values of move i and $i+1$ is -12), from Eqs.(14) and (15) one gets $13 \leq y[i+1]$ and $1 \geq y[i]$; in other words, the decision variables are fixed to $y[i+1]=13$ and $y[i]=1$, once $b[i,3]=1$. In a similar fashion, Eqs.(10)–(13) are strong enough to fix the difference of card values to $1(-1)$ once $b[i,1(2)]$ is set to 1.

Black Hole has no objective function, so we employ an artificial one, based upon a breadth-first strategy. This is because breadth-first is expected to yield a feasible solution easily. One such objective function is $\min \sum_{i=1,...,17} A.i.x[i] + \sum_{i=18,...,34} i.x[i]$, where A is a large constant: penalties for playing top row cards at a later stage in the game are higher, whereas there is no such penalty for bottom row cards. Preliminary experimentation confirmed that this objective function performed better than any other we considered.

7.2. MIP-2

The second model (MIP-2) proposed for the "Black Hole" problem is similar to MIP-1 in many ways. However, the most crucial difference is the use of implied constraints in a way that reduces the search space dramatically. Although the increase in the number of constraints has adverse effect on the speed of the solution algorithm, MIP-2 performs still better compared to MIP-1.

The underlying observation in developing MIP-2 is that at any stage of the game, one can infer about the possible card values for the next 11 stages. Consider Fig.(4) in which gray cells point to feasible card values for the beginning of the game.

Although Fig.(4) is useful on its own, it is possible to generalise it for any stage and for any card value to have more inference on the game. Fig.(5) is a result of such an effort and shows an instance of having a value of "6" at stage 15. Again the gray

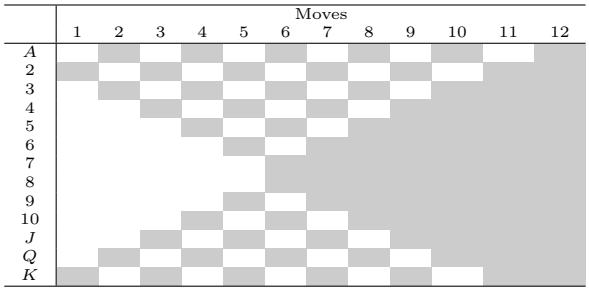

Fig. 4. Feasible card values for the beginning of the game

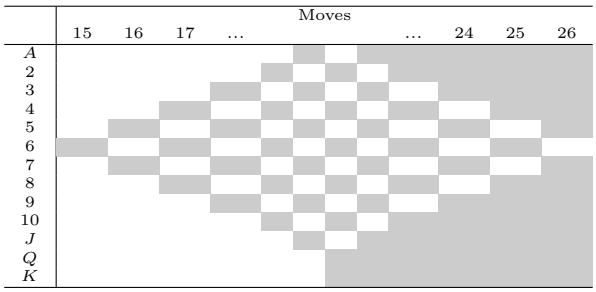

Fig. 5. Feasible card values for Value=6 at Move=15

cells represent all possible values if we have a "6" at stage 15.

In MIP, one way of expressing the case given in Fig.(5) is as follows:

$$\sum_{\substack{k\in\{1,...,51\}\\ v[k]=i}} a[k,j] + \sum_{\substack{k\in\{1,...,51\}\\ |v[k]-i|\neq\{h-1|h\in S_j\}}} a[k,j+m] \leq 1 \tag{18}$$

where, $j = 1, ..., 51$, $i \in S_j$, $m = 1, ..., \min\{11, 51-j\}$ and $S_1 = \{2, 13\}$, $S_2 = \{1, 3, 12\}$, $S_3 = \{2, 4, 11, 13\}$, $S_4 = \{1, 3, 5, 10, 12\}$, $S_5 = \{2, 4, 6, 9, 11, 13\}$, $S_6 = \{1, 3, 5, 7, 8, 10, 12\}$, $S_7 = \{2, 4, 6, 7, 8, 9, 11, 13\}$, $S_8 = \{1, 3, 5, 6, 7, 8, 9, 10, 12\}$, $S_9 = \{2, 4, 5, 6, 7, 8, 9, 10, 11, 13\}$, $S_{10} = \{1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$, $S_{11} = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$, $S_{12\leq} = \{1, ..., 13\}$

Eq. (18) exploits that, at any move, one can infer infeasible card values for the next 11 moves. Consider Fig. 4, in which gray cells are feasible card values for the beginning of the game. Fig. 4 can be generalised to any move and for any played card value. S_j denotes the set of feasible card values in move j. In Eq.(18), the first summation term gives all binary decision variables referring to playing card valued i in move j. The second includes all $a_{k,j+m}$ that don't comply with having a card valued i in move j. If any of the variables specified in the first summation takes on the value of 1, then all $a_{k,j+m}$ in the second must be 0.

MIP-2 consists of Eq.(18) as well as Eqs. (1) and (2) implying that card i can be played in move j only if the cards on top of it have been played, Eqs. (4) and (5) guaranteeing that only one card is played in each move, and each card is played only once.

7.3. MIP-3 and MIP-4

One weakness inherent in MIP is the relatively complex expressions needed for expressing lexicographical ordering. This has been observed in MIP-1 and MIP-2 models. In this section an alternative formulation is given for Eqs.(1) and (2).

The basic idea behind this formulation is the observation that if card i (assume that it is a middle row card) needs to be played at move j, then the top row card must have been played until stage j. This formulation provides us more inference compared to Eqs.(1) and (2). The downside of this alternative formulation is the large number of constraints it requires.

This alternative formulation, MIP-3, consists of $(i = 1, ..., 17)$

$$\sum_{k=1}^{j} a[i,k] \geq a[i+17, j+1], \qquad j = 1, ..., 50 \tag{19}$$

$$\sum_{k=1}^{j} a[i,k] \geq a[i+34, j+2], \qquad j = 1, ..., 49 \tag{20}$$

$$\sum_{k=1}^{j} a[i+17,k] \geq a[i+34, j+1], \quad j = 2, ..., 50 \tag{21}$$

as well as Eqs. (4), (5), and (18).

In MIP-4, lexicographical ordering constraints are expressed twice; first by means of Eqs.(1) and (2), and then Eqs.(19)–(21).

7.4. Computational Experience

To gauge the computational performance of these four MIP formulations we conducted numerical experiments. Experiments are performed on a 2GHz PC using the well-known MIP solver ILOG CPLEX 9.0. CPLEX's emphasis indicator is set to "emphasize feasibility over optimality". The test suite used in the experiments consists of 30 randomly-generated instances. The allowed maximum solution time is set to 5 hours. In three instances (19, 20, 29) there are no feasible solutions.

MIP-1 model contains 1071 constraints and 2908 variables (2805 binary variables). MIP-2 contains

	MIP-1	MIP-2	MIP-3	MIP-4
1	–	–	11000(1444)	8800(975)
2	–	–	7600(966)	800(29)
3	–	–	1800(52)	2300(85)
4	–	–	–	–
5	–	5600(1691)	850(71)	–
6	–	–	140(0)	2200(174)
7	–	–	300(7)	700(19)
8	–	–	–	–
9	–	–	1400(81)	–
10	–	–	2100(199)	–
11	–	–	970(21)	620(18)
12	–	1500(474)	330(15)	1400(100)
13	–	–	4700(415)	2600(148)
14	–	–	4400(414)	670(35)
15	–	–	–	–
16	–	710(168)	1300(82)	–
17	–	–	12000(1201)	–
18	–	2300(472)	670(22)	5100(235)
19*	–	–	38(0)	32(0)
20*	0.45(0)	0.42(0)	0.52(0)	0.54(0)
21	–	–	8500(933)	380(5)
22	–	–	–	–
23	–	–	–	17000(1396)
24	–	–	1300(143)	–
25	–	–	–	–
26	–	–	–	18000(1606)
27	–	–	1400(70)	1700(68)
28	–	–	470(5)	920(23)
29*	0.08(0)	0.42(0)	0.52(0)	0.55(0)
30	–	–	17000(2203)	–

Table 3

Solution time in secs (Number of nodes visited)
(* denotes "no solution" exists; A dash indicates no feasible solution in 5 hours)

6269 constraints and 2652 variables (2601 binary variables). MIP-3 and MIP-4 use 8683 and 8734 constraints, respectively, and 2601 variables.

It has been observed that MIP-3 is the computationally most efficient formulation. Using MIP-3, we were able to solve 23 instances out of 30. It is interesting to observe that in the solved cases the number of nodes visited are actually very small, pointing out that the MIP formulation is effective. The maximum number of nodes visited was 2203. However, the downside of the MIP approach is the excessive amount of processing time required at each node visited. The infeasible cases were easy to spot using MIP-3 or MIP-4. The infeasibilities were proven at the root nodes in less than a second.

These results suggest that IP is not an effective approach to address "Black Hole". This is interesting, since, as noted, IP has been successfully applied to many other fixed-length AI planning problems. The lack of a real objective function to provide a tight bound is certainly a factor, as well as the fact that the linear encoding of Black Hole requires a very large number of binary variables and constraints.

8. A Special Purpose Solver for Black Hole

Writing a special-purpose solver and encoding into another domain are both viable options for many classes of AI problems. The advantage of a special-purpose solver is that, knowing the properties of the problem, code can be optimised to search exceptionally fast. The disadvantage is the lack of mature and deep techniques for search, or the difficulty of adapting and implementing these techniques for the domain. In the case of Black Hole, we were able to write a special-purpose solver which could search very fast, but its lack of reasoning abilities means that the cost in larger search spaces is not repaid by the added speed per node compared to other methods.

The solver is written in Common Lisp. To avoid garbage collection, no lists were constructed or discarded after initialisation at the root of the search tree. The other key design principle was to minimise the amount of work on making moves and undoing them for backtracking. Indeed, there is only very moderate work done on choosing a move, and even less on backtracking.

Essentially we treat cards as pointers into the data structures. Each card is represented as an array index, so cards are numbered from 0 to $suits * ranks - 1$. For each card, we construct at the root a static pointer to the card immediately above it in its pile, or a null pointer if it is on top. A simple dynamic bitarray indicates whether each card has been played or not at this point in search, leading to one bit change on moving forward and backtracking. A card is available if it has not been played, but the card above it (if any) has been. Our data structures make this a very cheap test. With only four suits in the standard game, we simply test all possible cards of the right ranks to see if they are available. The test is performed for each card of rank one above and one below the current card in the hole: the list of cards for each rank is computed statically at the root. The alternative is to maintain lists of available cards of each rank, which in general will be much shorter than 4, the number of suits. However, in its simplest form this

leads to creating and reducings lists, and however it is done involves some work at each node and undoing it on backtracking. To avoid garbage collection we set up a 2-D array to store available moves at each depth, the dimensions being depth in search and twice the number of suits. At a new depth we insert the possible moves into this array. No work in this array needs to be done on backtracking, except decrementing the pointer to the current depth. Enumeration of the set of moves is implemented iteratively, but search itself is recursive, so the search function is called when we move to the next depth. So various bits and pieces go onto the function stack when this happens. However, the depth is only the number of cards.

We deal with conditional symmetries as follows. First, we distinguish between 'unit' cards, i.e. cards at the bottom of a pile or above only cards of the same rank, and other cards that we call 'general'. A card's unity or generality is determined statically at the root, so lists of general and unit cards are stored at the root. When finding playable cards, we consider at most one unit card of each rank, and none at all if there are any general cards of that rank. This deals with this kind of conditional symmetry almost without overhead. Our solver is able to search at almost 100,000 backtracks per second on a 2GHz PC. However, we are prone to exceptionally hard problems: one winnable instance took 1,055,774,437 backtracks and 11,701.38 secs. [3]

We adapted the solver to deal with general conditional symmetry, i.e. when two general cards of the same rank are simultaneously available. After backtracking from the choice of the first such card, it will be locked as unavailable until the card underneath the second card has been played and frees the first. In implementation, this freeing card is pointed to from the first, and the first card is not available until the freeing card has been played. When there are more than two cards available, the card underneath the third card will free the second, and so on. This array of freeing pointers does have to be maintained dynamically, but it is easy to calculate freeing cards from the list of available moves at each depth in search. The overheads are now more substantial, and we did not find great reductions in search from dealing with conditional symmetry, so overall performance was not dramatically improved.

To conclude, we did succeed in writing a solver to search very fast. Although our design decisions may not be optimal, we would be surprised if it could be speeded up to search 1,000 more nodes per second, as would be necessary to reduce the hardest problems to a few seconds each. Compared with the more intelligent and successful solvers reported earlier, we expect that some form of reasoning happening in those solvers is reducing the amount of search: duplicating this in a special-purpose solver would likely lead to the fastest possible search, but of course at a substantial overhead in programmer time.

[3] By mistake this random instance only involved 48 cards in 16 piles: but as claimed it worked correctly.

9. Experimental Evaluation

We have reported five solvers in this paper. We found that two, the MIP approach and the special purpose solver, were not competitive with the others, with many instances taking hours to solve. This in no sense implies a final conclusion that these approaches are impractical for Black Hole solving. However, the particular implementations we report here have not been the most successful, and we restrict their empirical evaluation to the brief details reported above. In particular, 30 instances using the MIP approach was enough to show that the cost per node was uncompetitive. Moreover, the special purpose solver may have become competitive if implemented in C/C++, with state-caching incorporated as for our CP solver. We have performed a much more extensive empirical comparison of our AI Planning, CP, and SAT based solvers. The CP solver results in this discussion are those for the version which incorporates state-caching, as described in Section 5.1.

We constructed a single benchmark set of 2,500 instances to test the solvers on. All three solvers were tested on the same instances, and they gave the same results on each instance – excepting a few timeouts described below. Since the instances were independently randomly generated instances of the standard game with a 52 card game, we can report on the expected winnability of Black Hole. A total of 2,189 instances were winnable and 311 were not, giving an 87.56% probability of winnability. The 95% confidence interval for the true probability is $[86.2\%, 88.8\%]$.

	FF	SAT	CP
> 2800s	43	0	0
100 – 999.9s	4	4	8
10 – 99.9s	359	160	61
1 – 9.9s	1,100	1,341	276
0.1 – 0.9s	643	770	648
< 0.1s	351	285	1,577
Max. sol.	130s	180s	510s
Median sol.	0.88s	1.75s	0.03s
Mean sol.	6.83s	3.47s	1.97s
Std. dev.	15.3s	7.6s	17.4s
win %	29.4%	7.7%	67.8%

Table 4

Solving 2,500 random instances of 'Black Hole': CPU time comparison.

We were not able to run the three remaining solvers on the same machines. Instead, we have normalised runtime results as if they were all run in the same machine, taking the CP solver to have a factor of 1. To derive the runtime multipliers we ran a single SAT solver on the same set of benchmarks on the three different machines. Our results are summarised in Table 4.

All three solvers were highly effective at solving these instances. Only the FF planner failed to solve all instances in less than 2,800s CPU time, and it solved all but 46, i.e. more than 98% of instances. Of those it did solve, the longest took only 130s to solve. For CP, the longest time was 510s, and for SAT the longest mean time was 180.5s (remembering that SAT times are means over a sample of runs).

This suggests a rank order of SAT, CP, FF. However, investigating percentiles of behaviour is more interesting. The best median is CP at only 0.03s, then FF at 0.88s and SAT at 1.75s. We see CP starting to outperform both FF and SAT at the higher percentiles. SAT solves 97.5% of instances in 17s, compared to 12s for CP and 65s for FF. Even this hides considerable complexity, as the three solvers find different particular instances difficult. There is only a weak correlation between the difficulty experienced by SAT and CP solvers, $r = 0.28$. The correlations between SAT and FF and between CP and FF are $r = 0.11$ and $r = 0.39$ respectively.

It seems clear that FF differs markedly from the other solvers, in that roughly 2% of our problems were found to be hard enough to time-out the FF computation, whereas all problems were solved (or proved unsoluble) in reasonable time by both SAT and CP. Moreover, mean performance of the SAT and CP solvers differs widely, at 3.47s and 1.97s respectively. This is distinct enough that a paired t-test on the two data sets rejects the null hypothesis that performance is the same ($p < 0.001$). We can therefore report that there is a statistically significant difference between the performances of the three approaches.

In practical terms, given our study of the profiles, it is reasonable to say that our CP solver is usually quicker, while the SAT solver is more robust at solving all instances in reasonable time. This is illustrated by the fact that the CP solver solves 88% of instances in less than 1s, while the SAT solver solves *all* instances in less than 3 minutes.

To conclude, we found that while all solvers could solve Black Hole instances, the SAT and CP solvers were the most effective. FF was just behind those two, with our other two approaches not nearly as successful.

10. Conclusions

We consider patiences in general, and Black Hole in particular, to be interesting domains for studying modelling and encoding, and analysis of empirical data. There are hundreds of patience games in existence, many of which will raise interesting questions which are dependent on the rules of the game in question. Moreover, people care about – and can easily understand differences between – random instances of the problems.

In this paper we have demonstrated that MIP, SAT encoding, AI planning, specialist solution and constraints are appropriate ways to study these games. Each of these AI methodologies uses modelling, encoding and solution techniques that differ from the others. By applying the methodologies to a problem that is both easy to understand and hard to solve (in general) we have been able to compare and contrast the relative efficacy of the methods. All the solution techniques examined in this paper are complete, since problem instances have a high probability of being both solvable and solvable in under one second. For other problem domains the use of local search search techniques should also be empirically evaluated.

The advantages of using games such as Black Hole are:

– any reasonably proficient AI practitioner can encode the game as an instance of their preferred methodology;
– the number of test cases is essentially unlimited (each being a shuffle of a deck of cards), so that the design and statistical analysis of experiments is straightforward;
– we can gain insights into the similarities and differences of competing AI methodologies;
– the well-understood framework makes it easy to disseminate results to the wider community;
– results can be applied to any fixed-length planning problem with perfect information that arises in the real world.

The scope for further work in this area is limitless: of the hundreds of patience games available, there are examples which incorporate perfect and imperfect information; allowed, limited and forbidden backtracking; multi-deck variations; and solution metrics such as shortest length. Each of these aspects is worthy of exploration in AI research.

Acknowledgements

The authors are supported by United Kingdom EPSRC grants GR/S30580/01 (Symmetry and Inference), EP/CS23229/1 (Critical Mass) and GR/S86037/01 (Symnet Network). Ian Miguel is also supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship.

References

[1] Allen, J., Hendler, J., and Tate, A. *Readings in Planning*. Morgan Kaufmann, 1990.

[2] Ansótegui, C., and Manyá, F. Mapping problems with finite-domain variables into problems with boolean variables. In *Seventh International Conference on Theory and Applications of Satisfiability Testing* (May 2004).

[3] Blum, A., and Furst, M. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)* (1995), pp. 1636–1642.

[4] Fish, S. Freecell solver. Web Page, 2002.

[5] Fox, M., and Long, D. PDDL2.1: An extension to PDDL for expressing temporal planning domains, 2001.

[6] Fox, M., and Long, D. Extending the exploitation of symmetries in planning. *Artificial Intelligence Planning Systems (Malik Ghallab, Joachim Hertzberg and Paolo Traverso, eds.)* (2002), 83–91.

[7] Gent, I. P., Kelsey, T., Linton, S., MacDonald, I., Miguel, I., and Smith, B. M. Conditional symmetry breaking. In van Beek [19], pp. 256–270.

[8] Gent, I. P., McDonald, I., and Smith, B. M. Conditional symmetry in the all-interval series problem. In *Proc. SymCon'03* (2003), pp. 55–65.

[9] Gent, I. P., and Prosser, P. SAT encodings of the stable marriage problem with ties and incomplete lists. In *Fifth International Symposium on Theory and Applications of Satisfiability Testing* (May 2002).

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

[11] Hnich, B., Smith, B. M., and Walsh, T. Models of permutation and injection problems. *Journal of Artificial Intelligence Research*, 21 (2004), 357–391.

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

[13] Jefferson, C., Miguel, A., Miguel, I., and Tarim, A. Modelling and solving English peg solitaire. *Computers and Operations Research 33*, 10 (2006), 2935–2959.

[14] Kautz, H., and Selman, B. Unifying SAT-based and graph-based planning. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence –IJCAI 99* (1999).

[15] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient SAT solver. In *DAC* (2001), ACM, pp. 530–535.

[16] Parlett, D. *The Penguin Book of Patience*. Penguin, 1990.

[17] Ryan, L. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, February 2004.

[18] Smith, B. M. Caching search states in permutation problems. In van Beek [19], pp. 637–651.

[19] van Beek, P., Ed. *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2003, Sitges, Spain, October 2005, Proceedings* (2005), vol. 3709 of *Lecture Notes in Computer Science*, Springer.