SEARCHING WITH ANALYSIS OF DEPENDENCIES IN A SOLITAIRE CARD GAME

B. Helmstetter, T. Cazenave
Universite Paris 8, laboratoire d'Intelligence Artificielle
2, rue de la Liberte 93526 Saint-Denis Cedex France
{bh, cazenave}@ai.univ-paris8.fr

Abstract We present a new method for taking advantage of the relative independence between parts of a single-player game. We describe an implementation for improving the search in a solitaire card game called *Gaps*. Considering the basic techniques, we show that a simple variant of *Gaps* can be solved by a straightforward depth-first search (DFS); turning to variants with a larger search space, we give an approximation of the winning chances using iterative sampling. Our new method was designed to make a complete search; it improves on DFS by grouping several positions in a *block*, and searching only on the boundaries of the blocks. A block is defined as a product of independent sequences. We describe precisely how to detect interactions between sequences and how to deal with them. The resulting algorithm may run ten times faster than DFS, depending on the degree of independence between the subgames.

Keywords: depth-first search, dependency-based search, block search, Gaps

1. Introduction

In this paper we consider a solitaire card game usually called *Gaps*, *Montana*, *Rangoon* or *Blue Moon*. We give approximations of winning chances for the game of Gaps and use the domain for testing new ideas. In the field of solitaire card games, we may also mention the game *Freecell* which has become a test domain in planning (Hoffmann, 2001).

We have reasons to believe that techniques based on heuristics are not very useful in Gaps. However we have been able to improve the search in another way, by proving the independence between moves in different parts of the game and making use of it. A few search techniques with similar concerns exist but they are based on different principles (Allis, 1994; Junghanns and Schaeffer, 2001; Botea, Müller, and Schaeffer, 2002). .

This paper is organised as follows. We explain the rules of Gaps in Section 2. We give results of the basic techniques and explain the reasons for our approach in Section 3. We present our method in Section 4 and experimental results in Section 5. Finally, in Section 6 we discuss possibilities for generalization and compare our method to the existing one which is the closest: dependency-based search (Allis, 1994).

After our experiments, it was a surprise to find that a game called *Superpuzz*, studied by Berliner and more recently by Shintani, is a variant of Gaps. However, at the time this paper was written, we did not know precisely what work has been done on Superpuzz. We have found a short description of Berliner's (1997) work on the web, and Shintani's (1999, 2000) work has only been published in Japanese.

2. Rules of the Game

Below we explain the rules of what we call the basic variant (2.1). Then we describe a few other variants (2.2). Finally, we give some basic properties of the game (2.3).

2.1 Basic Variant

The game is usually played with a 52-card deck. The cards are placed in 4 rows of 13 cards each. The 4 Aces are removed, resulting in 4 gaps in the position; then they are placed in a new column at the left in a fixed order (*e.g.*, 1st row Spade, 2nd Heart, 3rd Diamond, 4th Club). The goal is to create ordered sequences of the same suit, from Ace to King, in each row. A move consists in moving a card to a gap, thus moving the gap to where that card was. A gap can be filled only with the successor of the card on the left (that is, the card of the same colour and one higher in value), provided that there is no gap on the left and that the card on the left is not a King, in which case we can place no card in that gap. Figure 1 shows an initial position with only 4 cards per suit, before and after moving the Aces, and the possible moves.

4♢	3♣	4♣	4♡
2♠	4♠	2♡	2♣
2♢	3♠	1♡	1♣
3♢	1♠	3♡	1♢

1♠	4♢	3♣	4♣	4♡
1♡	2♠	4♠	2♡	2♣
1♢	2♢	3♠		
1♣	3♢		3♡	

Figure 1. An initial position with 4x4 cards, before and after moving the Aces. This position can be won.

2.2 Other Variants

The basic variant presented above is probably not the most common. Usually the Aces are not placed in a new column but are definitively removed. Instead it is allowed to place any Two in a gap if it is on the first column. This gives more possibilities than in the basic variant where only one Two can go in each place of the second column (which was the first before moving the Aces). This difference is not a minor one, as it has a strong influence both on the size of the search space and on the probability of a winning game, as we will see. We call this variant the common one. The reason for our choice of the basic variant was to make the rules cleaner; this way only one card can be placed in any gap.

The game is usually played with an additional rule which says that when the player gets stuck, he may remove the cards that are not part of an increasing sequence starting from the first column, and redeal them. Two redeals are allowed. We have not studied the game with this rule. However, as we will see, the probability of winning without this rule but with perfect play is higher than that obtained by human players using this rule.

It is possible to change the number of suits and the number of cards per suit. This influences the size of the search space and the problem's difficulty. It also has an effect on the degree of independence between subgames, which will be a major concern.

The game that has been studied under the name Superpuzz is what we have called the common variant. There is only one minor difference: the gaps are created by removing the Kings instead of the Aces.

2.3 Properties

In the basic variant, every initial dealing results in a separate game of perfect information. This version has a remarkable property: in any position, the depth of the search graph is bounded; in particular there is no cycle. If we look at a particular card, of value v, there is only $v-1$ places where it could be in the subsequent positions, in addition to its present location: it could be one space on the right of the card of the same suit and of value $v-1$, two spaces on the right of the card of the same suit and of value $v-2$ (which means that we have built a sequence $v-2, v-1, v$ from the current position of the card of value $v-2$ and of the same suit) $\ldots$, and $v-1$ spaces on the right of the Ace of the same suit. The card cannot go to any of those places twice, so the number of moves of this card is bounded by $v-1$. Therefore the total number of moves with 52 cards is bounded by $4 \times (1+2+3+\ldots+12) = 312$.

3. Basic Techniques

In this section we show results of either a depth-first search or an iterative sampling search applied to Gaps, then we discuss possibilities for improvement. Although the techniques exposed here are basic, the results are about the best we could do without using the method, block search, that we present in the next section.

3.1 Depth-First Search

It turns out that in the basic variant the search space is small enough to allow a complete depth-first search enhanced with a transposition table. Assuming that we stop the search as soon as we find a winning path, the average size of the search space is about 250,000. It seldom goes above 2 million. This is small enough for all positions to be stored in a transposition table. A test on 10,000 initial positions shows that the probability that an initial position can be won is about 24.8%. The length of winning solutions is usually in the range of 90 to 130 moves. All the computations have been made on an Athlon 1600+ with 1GB RAM. The previous search takes about 0.2s per problem.

This is only the beginning of the story though, because the basic variant is far from being the most difficult one, and even in the basic variant the difficulty could be increased by playing with more cards.

3.2 Iterative Sampling

DFS is impractical in variants where the size of the search space is too big. Instead, iterative sampling (Harvey and Ginsberg, 1995) has proved to be surprisingly efficient. This consists in playing completely random moves until a goal is found or the player gets stuck, in which case the search restarts at the beginning. This is repeated until a probe is successful or the maximal number of probes is reached.

We give results of this algorithm both for the basic variant and for the common one (where the Aces are definitely removed and any Two can go in the first column). We consider the common variant because the typical size of its search space is too big to allow a complete search in a reasonable time (this property could also have been obtained by increasing the number of cards). This way we also get a first approximation of the winning chances for the common variant, which are unexpectedly high.

Table 1 shows the results of an experiment on a set of 1000 random initial positions. The set of positions is always the same, except that, for accuracy, experiments with fewer than 1000 probes have been made on 100,000 initial positions. One probe takes about 4.5μs. This amounts to about 450 s for 10^8

probes when they are unsuccessful; on average 425 s and 164 s for the basic and common variants, respectively.

max number of probes	success rate basic variant	success rate common variant
1	0.046%	0.041%
10	0.373%	0.396%
10^2	1.37%	2.96%
10^4	7.1%	26.6%
10^5	9.7%	43.1%
10^6	12.8%	53.0%
10^7	14.5%	60.3%
10^8	16.4%	66.9%

Table 1. Results of iterative sampling.

It is a particularity of our domain that we have a slight chance of winning by doing random moves. The efficiency of the algorithm is due to its covering a well-distributed part of the search space and its avoiding getting lost in large parts of the tree where it is impossible to win.

3.3 Combining a Depth-bounded Search with Iterative Sampling

Iterative sampling can be combined with a depth-bounded complete search. One possibility is to make a breadth-first search until exhaustion of memory resources, and to make one or more random probes at each node of this search. The results are better compared to simple iterative sampling, probably because it ensures a better distribution of the probes in the search space. Furthermore this method will also prove some problems impossible when the search space is searched completely.

We have run a test on 100 random initial positions for the common variant. The breadth-first search was limited by the number of positions that could be stored in memory: this number was set to $5,000,000$. One random sample was performed at each node. The program took 144 s per problem in average, and it has found solutions for 88 of the initial positions and proved 4 impossible.

3.4 Comparison with Human Performance

Estimations of the chances of winning for human players are based on various sources from the web and on personal experience. The chances of winning when no redeal is allowed are of about 1%. The exact rule concerning the gaps in the first column apparently has little effect on the difficulty of the game for human players, but we have shown it is important for the computer. The last feature must be compared with 24.8% (basic variant, complete DFS), 66.9% (common variant, iterative sampling) and 88% (common variant, combination of breadth-first search and iterative sampling).

With two redeals allowed, chances of winning for human players are of about 25%, still well below 88%.

3.5 Discussion on Possibilities for Improvement

Iterative sampling is a simple and efficient technique for the game of Gaps; however, in the process of designing a search program for solving Gaps, this took us quite some time to realize. Previously, our attempts for solving Gaps and its variants were based on complex best-first search algorithms. We gradually realized that it was important to try and go to the goal often without caring much for the quality of the moves. At that time our algorithms were about as follows: do some tree search in a best-first way, and during this search, from time to time, launch random samplings. We finally found that the strength of our program was almost entirely due to the random samplings.

At the beginning, we had been working on yet another variant of the game. This variant differs from the common one by the rule that we may move in a gap not only the successor of the card on the left but also the predecessor of the card on the right. We felt that there were much more efficient heuristics in this variant. We did get some successes using heuristics, but even there random sampling alone would do about as well.

In the basic variant the situation is worse. What heuristics do we have? First, there is the number of cards that are already in their final location. This is the only simple heuristic we know about, but unfortunately it gives a poor evaluation of the position, as it often happens that most cards only get in their correct location in the endgame. Then there could be heuristics concerning the mobility of the cards, in the present and in the long term, but this is difficult to estimate.

It is possible that good heuristics could be found. However a comparison with human performance shows that we are not so bad with iterative sampling. One can see that one sample by human players is roughly as successful as 100 random samples. This indicates that human players do not use very efficient heuristics anyway. Even if we could do as well as humans on one sample, considering the time that would be needed for computing heuristics it would probably not be interesting. Because heuristics are weak, any best-first search algorithm would also be of limited use. As an example there is the well-known IDA*; we have experimented with it but did not achieve better results than with a depth-first search.

The next part of this paper takes an orthogonal approach to the heuristic one. Our goal is to go through the search space completely, without even caring whether we find a winning sequence. We want to do it faster than a depth-first search would, by simplifying the search space. Thanks to this, we will be able to determine for sure if there is a solution in some problems where depth-first search is not applicable; for instance in the basic variant when playing with more cards.

4. Block Search

In this section we present a new method that aims at proving the independence between some parts of the problem and taking advantage of it, while keeping the search complete.

From now on the focus will be on the basic variant only, because we will make use of the fact that in any position only one card can be placed in a gap. The common variant does not have this property, and therefore more work would be needed in order to generalize the method to this variant.

4.1 Related Work

Among existing search algorithms, the closest to ours is probably the algorithm dependency-based search by Victor Allis (1994). He applied his method to three domains: the double letter problem, and the search of winning sequences in Qubic and GoMoku (the last two, being 2-person games, were first transformed into single agent games). In fact the starting point of our work was a failure to adapt this algorithm to the game of Gaps. A pseudo-code for the algorithm was given, but a function called *NotInConflict* was not explicit; we believe that this function was easy to write in the domains where the algorithm had been implemented but would be difficult to write in Gaps, at least not statically.

The goal of our method is also similar to that of Junghanns' Relevance Cuts for Sokoban (Junghanns and Schaeffer, 2001). He suggested that relevance can be approximated by computing an *influence* between moves, and then penalizing moves that are not relevant to the previous ones. His work was done in the context of an IDA* search, so in his method moves are never definitely eliminated, they may only get a penalty. The method we have developed handles the problem more precisely.

A more recent work on Sokoban (Botea, Müller, and Schaeffer, 2002) addresses the problem in yet another way, by decomposing the position in rooms and precomputing the graph of states in each room. The major difference with our work is that the states in the subgames are precomputed, and this does not seem possible in Gaps.

4.2 Principle of the Method

We name the four gaps A, B, C, D, and break the game into four subgames also named A, B, C, D. The moves allowed in one subgame are those that use the corresponding gap. If one plays only in one subgame, one makes a linear sequence of moves. This sequence moves the same gap from place to place until getting stuck, which can happen for any of the following two reasons: either there is another gap on the left, or the card on the left is a King. Whereas the moves in the same subgame are totally ordered, moves from different subgames are often independent. We want to take advantage of this relative independence between the subgames.

Let a *block* be defined by its starting position, and in each subgame X a sequence S_X, possibly empty, from the starting position, so that the four sequences are independent of each other. A block represents a set of positions: all positions that can be reached from the starting position of the block with the moves of the sequences S_X in any order. It helps to see a block geometrically as embedded in a four dimensional space. If l_X is the length of sequence S_X, the block represents $l_A \times l_B \times l_C \times l_D$ positions. Finally, we call F_X the face of the block that consists in the positions of the block where all the moves of sequences S_X have been made.

The main idea of the algorithm is: instead of searching one position at a time, we search one block at a time. Instead of recursively searching the immediate children of a position, we construct new blocks at the boundary of a block and recursively search them.

We want to construct blocks of the biggest possible size, so before building blocks on the boundary, we try to extend them as much as possible in the four subgames. Figure 2 shows a pseudo-code for the algorithm.

void search(block) {
  for each subgame X
    extend block in the subgame X, as long as
    all the sequences keep being independent;
  for each subgame X
    build new blocks near the face F_X of the block,
    such that any move we can do from F_X goes to
    one of those blocks, and search them recursively;
  test for a winning position in the block;
}

Figure 2. Pseudo-code for block search.

We still have to show how to extend blocks and construct new blocks at the boundaries. Besides, the pseudo-code does not include a transposition table, and this will lead to some problems to be addressed in Subsection 4.7.

4.3 Study of the Basic Interaction

We study in detail the case of a single interaction between two sequences. Figure 3 shows the useful part of the position and a diagram which synthesizes the relation between the two sequences. For simplicity all cards are of the same suit. We assume that both sequences begin a few moves before the interaction and end a few moves after, although the moves that are not critical have not been drawn. The dotted arrow in the right diagram indicates the action of sequence S_B on sequence S_A. Assume we are just after move a_1 in S_A. If b_1 has not yet been made, move a_2 can be made in subgame A; if move b_2 has already been made, move a'_2 can be made in subgame A; if move b_1 has been made but not move b_2, no move can be made in subgame A.

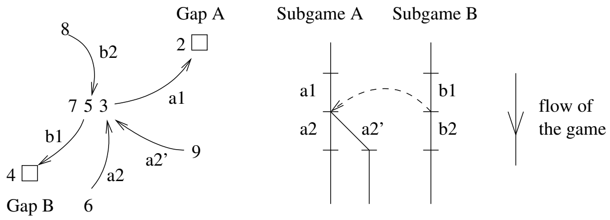

Figure 3. A basic interaction.

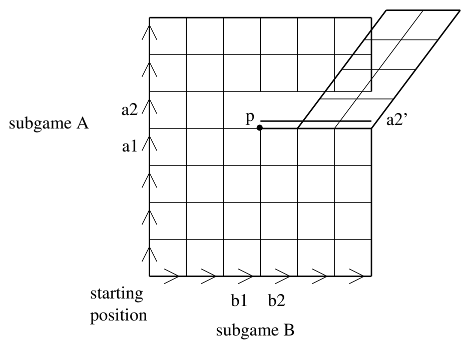

Figure 4. Search space corresponding to a basic interaction.

Figure 4 shows a representation in the plane of the search space, where the positions lie at the intersections of the lines. Imagine there was no interaction between the two sequences; we would have a big square with the entire sequences A and B on the edges. The effect of the interaction is to cut this square along a line from the point p to the right side (the double line in the figure), and to stick another part along the cut, which corresponds to sequence S_A taking the bifurcation. The position at p is particular: it is the position where the two gaps are adjacent, so that no move can be made in subgame A. This point corresponds to the dotted arrow in Figure 3. We say that there is a bifurcation of sequence S_A, caused by an action of sequence S_B. One must imagine that there are two other dimensions corresponding to the subgames C and D; if the sequences in these subgames do not introduce new interactions, the complete search space will be a simple product of the graph in Figure 3 with the sequences S_C and S_D.

We are looking for ways to partition the search space into blocks. There are several ways to do it; Figure 5 shows the two ways we will use. They deal with the two possible shapes of the first block. Whether we get in the first or in the second depends on the order in which we have extended the first block: first in subgame A or B. Subsection 4.6 will explain precisely how to detect interactions when extending blocks and how to build new blocks at the boundary. For the moment, we note that in the first possibility blocks 2 and 3 are children of block 1, whereas in the second only blocks 2 and 4 are children of block 1, and block 3 is a child of block 2.

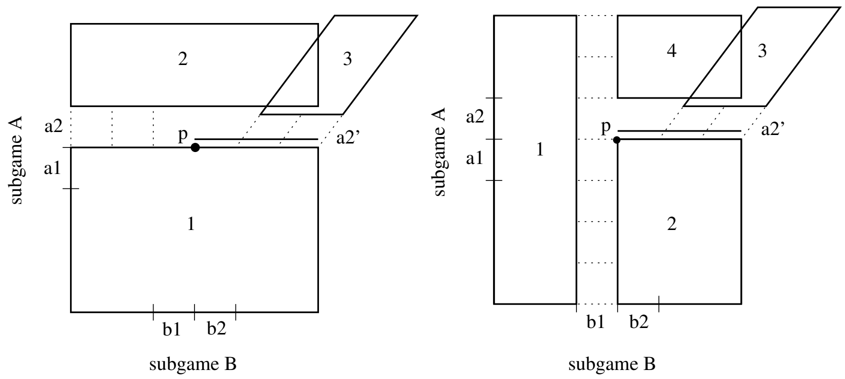

Figure 5. Two ways to decompose the search space into blocks.

4.4 Why the Basic Interaction is the Only One to Consider

We have shown how to deal with the basic interaction; it turns out that it is the only one we have to consider. Let S_X and S_Y be two sequences in the subgames X and Y. Let us enumerate all the ways that a move y of sequence S_Y could be influencing a move x of sequence S_X. Move x consists in taking the card c from place p_1 to place p_2. The prerequisites for this move are:

1 there is a gap at p_2,
2 the card c is in p_1,
3 the card at the left of p_2, c_L, is the predecessor of c.

Those preconditions are verified if we make only moves from sequence S_X up to x, but they could be destroyed by moves of sequence S_Y. Assume we have already established the independence of the sequences S_X and S_Y up to the moves x and y; then precondition 1 is automatically satisfied as soon as we have executed all the moves of sequence S_X up to x and whatever we have done in the sequence S_Y up to y, since it is an effect of the beginning of sequence S_X to put a gap in p_1.

Precondition 2 is automatically satisfied too. This comes from the fact that the card c can only go to the right of c_L, wherever this card be. If this card was moved by sequence S_Y, then there would already be an interaction because of precondition 3. If move y moved card c to the right of c_L and this card was still at the left of p_2, the trajectories of the gaps X and Y in the sequences S_X and S_Y up to x and y would both pass through p_2, which again would imply that they are dependent.

Therefore only precondition 3 remains to consider, which produces an interaction of the kind already analysed.

4.5 Using a Trace to Speed Up the Discovery of the Interactions

The *trace* of the sequences S_X is an array of the same size as the position (4x13). It contains information for each place p indicating whether and at which move the trajectory of the gap for any sequence is passing through p. The trace is maintained incrementally as we build new blocks.

Assume we make a new move m, which moves a card from p_1 to p_2. We want to know if it produces new interactions with the sequences already built. A look in the trace at the place just on the left of p_2 shows if any sequence has an effect on move m. A look in the trace at the place just on the right of p_1 shows if move m has an effect on any sequence. This way the search for an interaction is done very quickly.

The method of doing a local search and storing the set of properties of the position on which the result relies has already been used by other people in different domains: in Go, with the goal of incrementally updating local results (Bouzy, 1997); in Generalized Threats Search (Cazenave, 2002) which is a 2-players selective search algorithm that relies on a trace to find a set of relevant moves; in the algorithm H-search used in the hex program HEXY (Anshelevich, 2002) with a bottom-up approach, building increasingly complex virtual connections.

4.6 Building and Extending Blocks

The procedure for building blocks at the boundary is tricky, because we have to take into account all the interactions that might occur. Although there is only one kind of interaction that needs to be considered, it can come in the two different configurations shown in Figure 5, and we must be prepared that several configurations occur at a time. Figure 6 represents a search space in two dimensions that gives an idea of the kind of situations we have to deal with.

We are on face F_B of block b. We try to find out what moves can be made in subgame B depending on the exact location on F_B, and if the moves have an action on the other sequences. This situation occurs twice in the program: first when we are trying to extend the block in subgame B (which can be done as long as there is no interaction), second when we are building new blocks near face F_B (generally because we have already found an interaction). We must answer the following questions in this order.

1 Is there an action of any other sequence of the block that will cause a bifurcation on S_B? This is the case if and only if the trajectory of the gap for any other sequence is passing through the place at the left of gap B. This can be decided quickly by looking at the trace. An example is interaction 1 in Figure 6.

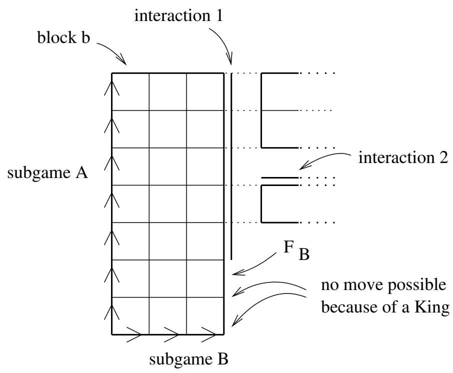

Figure 6. Several interactions on one face of a block.

2 Now we know that the move to be made in subgame B does not depend on the location on face F_B, or we have already restricted ourselves to an area where it is the case. Is a move possible at all? There cannot be another gap on the left of gap B because of the previous step, so the question is whether the card on the left is a King (in this case the block cannot be extended, and no block will be constructed on this part of the boundary).

3 Now we know we can do a move; this move takes a card from a place p and moves it in gap B. Is there an action of this move on the other sequences? This is the case if and only if the trajectory of the gap for any other sequence is passing through the place at the right of p. This too can be decided quickly by looking at the trace. An example is interaction 2 in Figure 6.

When building new blocks at the boundary, one must go through these three steps. In steps 1 and 3 we may have to break face F_B into two parts (in some degenerated cases there may be one or zero part) and apply the following steps to each. After step 3 we have isolated a part G of the face F_X. We know that we can make a move m anywhere on G and that this move has no effect on the other sequences. We then create a new block by doing move m from G, and search it recursively.

A block c that has just been built on face F_X of a block b has no depth in subgame X. When we try to extend block c in all the subgames, it is generally successful for subgame S_X; on the contrary, it is generally not successful in the other subgames because the reasons why block b had been stopped in those subgames often stand for block c.

4.7 Adding a Transposition Table

The decomposition in blocks already handles all the transpositions within the blocks; this is good but insufficient. In order for the algorithm to be efficient, it is almost compulsory to have a global transposition table. However, when we construct a new block or when we extend one, there could be common positions anywhere in this block and in another already built one.

Now we do not want to go to all the positions in every block and mark them in the transposition table, because the advantages of our method would disappear. We have to look for a compromise: we could mark only a well-chosen part of each block and hope it will be sufficient to detect most transpositions.

We define the number of positions of a block search as the sum, for each block, of the number of positions contained in this block. Let R be the ratio between the number of positions of a block search and the number of positions of a depth-first search. Ideally, if the transpositions table is large enough to contain all the positions of the search space and if the blocks are mutually disjoint, then $R = 1$. If the blocks are not mutually disjoint, then R will be larger; we need to control how much larger it will be.

A first possibility is to mark only the starting position of each block. An experiment on 100 random positions for the basic variant has shown that the ratio R is about 3.95, which is too much.

A second possibility is to mark only the positions that can be reached from the initial position of the block by making moves in only one of the four sequences of the block simultaneously. Geometrically, those are the points located on the four edges of the block starting from the initial position. The ratio R drops down to 1.33, which is acceptable although a better compromise probably exists.

5. Experimental Results

The method was designed to be complete; we have verified experimentally that it is indeed the case. This is a sign that we have correctly analysed all the possible interactions that can occur at the boundaries of the blocks. The method for verifying the completeness was the following: from an initial position, first run a complete depth-first search and store all the positions of the search space; then run a block search and verify that all the positions of the search space lie in at least one of the blocks. This verification has been done for 1000 initial positions.

Table 2 shows statistics about an experiment on 1000 random initial positions for the basic variant. There is a difference in time and number of positions compared to Subsection 3.1 because the search is not stopped when we find a winning position. Also the transposition table is not implemented in the same way: before it could grow as needed, now we use a hashtable of fixed size as is usual in game programming (Breuker, 1998). The hashtable has 64 million entries.

avg. number of positions for DFS	502,000
avg. number of blocks	36,200
avg. size of blocks	18.6
R	1.34
avg. time for DFS	2.28s
avg. time for block search	2.04s

Table 2. Basic variant (4 suits, 13 cards/suit).

The average size of the blocks is 18.6, so one node of block search does as much work as 18.6 nodes of DFS in average. As we have already mentioned, the total number of positions in all the blocks is larger than the number of positions searched by DFS, by about 34%. The final result in speed is a gain of 11% for blocks search. In the present case however, the difference in time is not very significant of the performance of block search because, for both algorithms, much time is spent reinitializing the large transposition table between problems.

We do not see the power of block search yet. Higher gains in speed can be obtained in variants with a larger search space, and with a higher degree of independence between the subgames. This can be achieved by increasing the number of cards. We therefore turn to 6 suits and 13 cards per suit. This increases both the number of cards and the number of subgames.

number of positions for DFS	289×10^6
number of blocks	5.00×10^6
size of blocks	59.7
R	1.03
time for DFS	437s
time for block search	44s

Table 3. 6 suits, 13 cards/suit, the hashtable has 64 million entries.

It is difficult to give average statistics because the size of the problems vary a great deal, some being too big for DFS and a few even for block search. We have made 15 experiments with initial positions that could be completely searched both with blocks search and DFS. In Table 3 we show detailed statistics for one of them, which is typical. We also show in Figure 7 that the gain in speed is correlated to the size of the search space. This

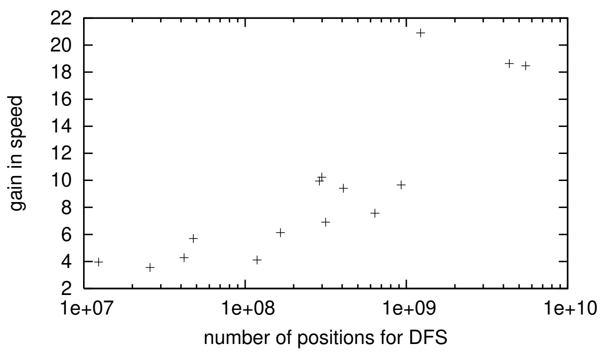

Figure 7. Gain in speed of block search over DFS, for 15 initial positions.

is promising. Our algorithm clearly has an advantage over a depth-first search because it can build large blocks, and this advantage would grow larger if the number of suits and/or cards per suit was further increased.

number of positions for DFS	1550×10^6
number of blocks	5.88×10^6
size of blocks	61.1
R	0.23
time for DFS	1730s
time for block search	46s

Table 4. 6 suits, 13 cards/suit,the hashtable has 8 million entries.

In the experiment of Table 3, the ratio R has dropped down from 1.34 to 1.03. This is due to collisions in the transposition table. This phenomenon is amplified if we decrease the size of the transposition table: Table 4 shows statistics about the same problem but with a hashtable that can only contain 8M positions. This has a dramatic effect on DFS, but almost no effect on our algorithm. Because of collisions, the ratio R is even less than 1! So, now the situation is reversed: it is our algorithm that makes a better use of the transposition table.

6. Perspectives

We conclude the paper by providing two perspectives. In Subsection 6.1 possibilities for generalization are given. In Subsection 6.2 dependency-based search is compared to block search.

6.1 Possibilities for Generalization

The general idea of the method does not rely much on the domain of Gaps. Our notion of a block can in principle find equivalents in many domains, provided that we generalize it a little. Until now we have worked with blocks that are products of independent sequences; as a first generalization, we should define blocks as products of independent graphs. In most domains there will be parts of the problem that will be, at least locally, relatively independent.

To apply the method, we must define what a subgame is, by stating which moves belong to which subgame, and we must analyse precisely all kind of interactions that could occur between them. This analysis is difficult and is domain-dependent, but then the rest is similar to what we did in Gaps: build and extend subgraphs in each subgame only as long as they keep being independent. The product of those graphs gives a block. Then we build new blocks at the boundary of this block and search them recursively.

Therefore we claim that the idea of decomposing the search space in blocks is a natural way to simplify the search space and may be applicable to other domains. Furthermore, the method could be much more powerful in domains with more independence between subproblems, leading to the construction of much larger blocks.

6.2 Comparison between Dependency-based Search and Block Search

As a first application domain to dependency-based search, Allis (1994) considered the double letter problem. In this domain, a state consists in a word on the set of 5 letters $\{a, b, c, d, e\}$. Any double occurrence of a letter can be replaced by a single instance of its alphabetical predecessor or successor. The alphabet is cyclic so ee can be replaced either by d or a, aa by either e or b. The winning states are the one-letter words. A detailed application of the method had been shown by Allis for the initial state $aaccadd$. A solution exists (the letters that change have been capitalized):

$$aaccadd \rightarrow Bccadd \rightarrow bBadd \rightarrow Aadd \rightarrow Edd \rightarrow eE \rightarrow A$$

We are going to compare the way this instance is solved by dependency-based search (according to Allis) and a way it could be solved by a block search algorithm. Dependency-based search runs with a succession of dependency stages and combination stages. After one dependency stage and one combination stage, he gets the graph in Figure 8: he considers the 6 moves possible at the root and finds that two can be combined together. The same situation can be represented with blocks (Figure 9): we have 3 independent subgames corresponding to the letters at the positions $1, 2$, $3, 4$ and $6, 7$, respectively. In each of those subgames two moves can be made from the initial position. Therefore the set of positions reachable with these moves can be represented with a cube, the initial position $aaccadd$ being in the centre. We then find an interaction at one of the edges of the cube: the two "B" that have been created allow to move in a new subgame and therefore a new block can be constructed.

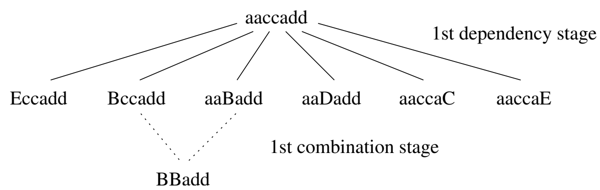

Figure 8. Dependency-based search, beginning.

The rest of the search continues similarly with dependency-based search (Figure 10) and block search (Figure 11). At least in this example we are really doing the same thing with different representations.

This goes to show that both methods have similarities. However, there are some differences that cannot easily be seen on the last example. First we do not see all the power of block search here: comparatively to dependency-based search, we believe it can deal with interactions of a more complicated nature (as in Gaps where we could not apply dependency-based search). Probably we

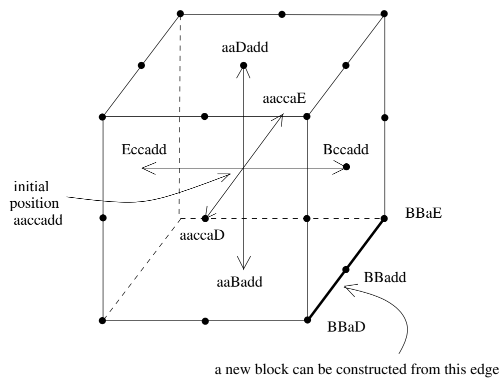

Figure 9. Block search, beginning.

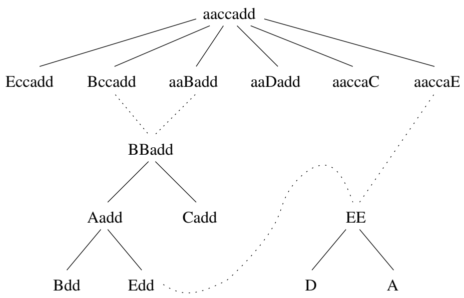

Figure 10. Dependency-based search, complete.

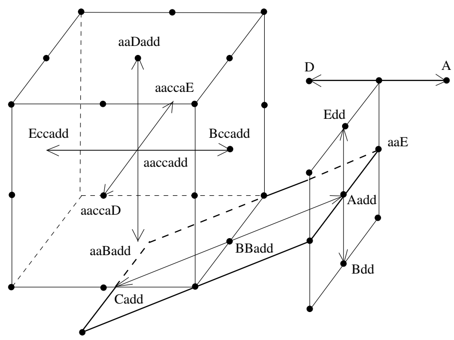

Figure 11. Block search, complete.

do not see all the power of dependency-based search either. For instance and in contrary to block search, it is not necessary to provide an explicit decomposition in subgames to apply dependency-based search.

7. Conclusion

We have presented several search algorithms that take advantage of the particularities of the game of Gaps. Our work has resulted in a method, block search, which may be applicable in other domains.

We have shown that iterative sampling produces good results, either for the basic variant or the common variant. Conversely, we have shown that the use of heuristics is not so promising. Therefore we could deal with only one problem in isolation: exploiting the independence between parts of the game. Existing methods that deal with this problem were either not applicable to the domain of Gaps or were not as precise as ours.

Block search is a method to take advantage of a decomposition in subgames when there are interactions between those subgames, while keeping the search complete. It implies analysing theoretically all types of interactions that can occur: how to detect them, how to deal with them by building new blocks at the boundary of the current block. Although this analysis relies on domain-dependent knowledge, the general idea of the method does not. Experimental results have shown that large gains in speed over a depth-first search can be expected, depending on the average size of the blocks we are able to build. Specifically, the method can be used to solve positions of the basic variant of Gaps with more cards. Because the method simplifies the search space, it also makes better use of a transposition table.

References

Allis, L. V. (1994). Searching for solutions in games and artificial intelligence. Ph.D. thesis, Rijksuniversiteit Limburg, Maastricht.

Anshelevich, V. V. (2002). A hierarchical approach to computer hex. *Artificial Intelligence*, Vol. 134, Nos. 1-2, pp. 101-120.

Berliner, H. (1997). Research interests. http://www-2.cs.cmu.edu/∼burks/frg97-98.html.

Botea, A., Muller, M., and Schaeffer, J. (2002). Using abstraction for planning in sokoban. In *Proceedings of Computers and Games*, Edmonton, Canada.

Bouzy, B. (1997). Incremental updating of objects in indigo. In *Proceedings of the Fourth Game Programming Workshop in Japan*.

Breuker, D. M. (1998). *Memory versus search in games*. Ph.D. thesis, Universiteit Maastricht, Maastricht.

Cazenave, T. (2002). A generalized threats search algorithm. In *Proceedings of Computers and Games*, Edmonton, Alberta.

Harvey, W. D. and Ginsberg, M. L. (1995). Limited discrepancy search. In *Proceedings of IJCAI*, Vol. 1, pages 607-615.

Hoffmann, J. (2001). Local search topology in planning benchmarks: An empirical analysis. In *Proceedings of IJCAI*, pages 453-458.

Junghanns, A. and Schaeffer, J. (2001). Sokoban: improving the search with relevance cuts. *Theoretical Computer Science*, Vol. 252, Nos. 1-2, pp. 151-175.

Shintani, T. (1999). A Program to solve Superpuzz. In *Game Programming Workshop in Japan '99*, pages 84-91, Kanagawa, Japan. (In Japanese)

Shintani, T. (2000). Consideration about state transition in Superpuzz. In *Information Processing Society of Japan Meeting*, pp. 41-48. Shonan, Japan. (In Japanese)