Dynamic Decomposition Search: A Divide and Conquer Approach and its Application to the One-Eye Problem in Go

Akihiro Kishimoto
Department of Computing Science
University of Alberta
Edmonton, Canada, T6G 2E8
kishi@cs.ualberta.ca

Martin Müller
Department of Computing Science
University of Alberta
Edmonton, Canada, T6G 2E8
mmueller@cs.ualberta.ca

Abstract- Decomposition search is a divide and conquer approach that splits a game position into sub-positions and computes the global outcome by combining results of local searches. This approach has been shown to be successful to play endgames in the game of Go. This paper introduces *dynamic decomposition search* as a way of splitting a problem dynamically during search. Our results in solving one-eye problems in the game of Go show the promise of this approach. Additionally, we propose *relaxed decomposition*, a more ambitious way of splitting positions.

1 Introduction

In two-player games with perfect information, programs typically employ lookahead search to determine which move to play. Traditional brute-force search algorithms explore all possible moves as deeply as possible in order to improve the strength of a program. Such approaches have been very successful, enough to reach the strength of the best human players in games such as chess and checkers [2, 8]. However, the game of Go has been resistant to a pure search-based approach because of its difficult position evaluation and large search space. Current computer Go programs use a combination of exact and heuristic rules instead of brute-force search.

One clear defect of heuristic approximations is that they sometimes fail. As a result, all current computer Go programs show weaknesses in assessing the life and death status of groups, the so-called *tsume-Go* problem. In general, search is the only way to reliably determine the life and death status of stones. Therefore we need to find effective ways to tackle the large branching factor of Go.

One approach to overcome the large search space is to divide a position into independent subpositions and combine the outcome of local searches. This way, we can not only achieve a large reduction of the search space, but also guarantee correctness. *Decomposition search* is such a divide and conquer approach for Go endgames [5]. Using this approach, programs can solve a much larger class of endgame problems than with classical minimax-based solvers. However, in basic decomposition search a problem is split into subproblems only at the root node of a search. There are no further splits during the search, so the method fails for example when there is just a single large undivided area in the beginning. In applications such as tsume-Go, it seems necessary to do decomposition *dynamically* within the search tree.

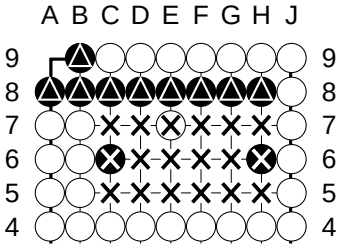

Figure 1: Example of a one-eye problem (Black to live).

This paper presents a dynamic divide and conquer approach to the problem of making *one eye* in an enclosed region. Experimental results show that our approach achieves better performance on average. Also, we introduce the notion of *relaxed decomposition*, which achieves a more fine-grained division into sub-problems.

The structure of the paper is as follows: Section 2 briefly describes the one-eye problem in Go. Section 3 reviews previous work. Section 4 presents the dynamic decomposition algorithm. Section 5 discusses empirical results. Section 6 introduces relaxed decomposition search, and Section 7 presents some conclusions and discusses future work.

2 The One-Eye Problem in Tsume-Go

The one-eye problem in Go is a special case of Life and Death (tsume-Go). It addresses the question of whether a player can create an eye connected to the player's stones in a given region. A problem can be investigated for either player moving first. Although this problem is simpler than full tsume-Go, which is concerned with making two eyes, there are many similarities. For example, every tsume-Go problem in which the group under attack has already surrounded one eye in some region reduces to the one-eye problem on the rest of the board.

A one-eye problem in a given Go position is defined by the following input:

- The two players, called the *defender* and the *attacker*. The defender tries to make an eye and the attacker tries to prevent it.
- The *region*, a subset of the board. At each turn, a player must either make a legal move within the region or pass.

- One or more blocks of *crucial stones* of the defender. The defender wins a one-eye problem by creating an eye connected to all the crucial stones inside the region. The attacker can win by either capturing at least one crucial stone, or by preventing the defender from creating a connected eye in the region.
- *Safe attacker stones*, which surround the region together with crucial defender stones.

In the remainder of this paper, whenever we briefly say "make one eye" we mean "create an eye connected to all the crucial stones inside the region", as above.

Figure 1 shows an example of a one-eye problem. Black is the defender and White is the attacker. Crucial stones are marked by triangles and the region is marked by crosses. Black must make an eye inside the region, while White tries to prevent that. There are unsafe stones at **C6**, **E7**, and **H6**. If these stones are captured, a player might play at such a point later, so they are part of the region.

3 Related Work

3.1 Decomposition Search

In decomposition search, a position is split into subpositions, which are surrounded by safe stones [5]. Local searches, based on combinatorial game theory [1], are then used to analyze each subposition. If all subproblems have loop-free values, then the combinatorial game values of the subproblems can be combined to achieve globally optimal play. As mentioned above, all decomposition happens at the root and there is no further decomposition during search.

3.2 Previous Work on Our One-Eye Solver

The previous version of our one-eye solver is described in [3]. It uses a modified version of Nagai's depth-first proof-number (df-pn) search algorithm [7] which can deal with ko repetitions. The solver checks each point in a given region to find all *potential eye points*. If a complete eye connected to crucial stones is found, the defender wins. If no potential eye point exists, the attacker wins. In all other cases, df-pn search is performed. The basic solver generates all legal moves including a pass. The solver therefore guarantees correctness for enclosed positions. An improved version adds some simple and correct Go-specific knowledge, such as detecting connections to safe stones and forced moves that safely prune other moves. Despite a relatively small amount of Go-specific knowledge, this solver succeeded in solving hard problems that are not solved by the best general tsume-Go solvers in a reasonable time.

3.3 Related Work on Tsume-Go

Wolf's *GoTools* is currently the best tsume-Go solver that specializes in solving completely enclosed positions [10]. GoTools contains a sophisticated evaluation function that includes look-ahead aspects, powerful rules for life and death recognition, and learning dynamic move ordering from the search [11]. Most competitive Go programs also contain a tsume-Go module. The commercial database *Tsume-Go Goliath* uses a proof-number search engine to check the user's inputs. Vilà and Cazenave presented a static approach to detect large eye shapes [9]. Such eye shapes guarantee life by either dividing it into two eyes or living in seki.

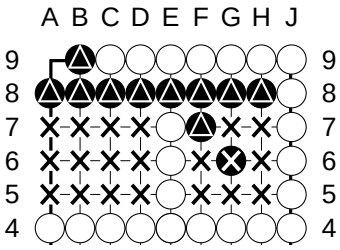

Figure 2: A position to which a divide and conquer approach is applicable. (Black to play.)

4 Dynamic Decomposition Search for the One-Eye Problem

4.1 Basic Idea

The basic idea of our divide and conquer approach is simple. For example, assume that Black needs to make the second eye in Figure 2. A naive algorithm would generate moves at all marked points in its search. This is clearly inefficient, since the marked region is already split into two separate areas. With the exception of *ko* fights, no move played in one area can affect the result of whether there is an eye in the other area. Instead of performing a global search, a divide and conquer approach performs two local searches that can be combined into a global result. This approach can reduce the branching factor and depth of the search by a large margin.

However, if ko fights are involved in a local solution, this approach can change the ko status because formerly local ko threats become non-local. Figure 3 presents such a case. In this example, White can capture the ko first but Black has a local ko threat at 2. White needs one external ko threat to win. However, if the region is divided into two parts, and the solver looks only for eyes, it will miss the ko threat at 2 in the other region, and the ko becomes one where Black needs an external ko threat.

To fix this problem, the solver would need to be extended to search for ko threats in all subregions whenever the result of the one eye search is a ko. This is currently not implemented. There are further complications, for example if two or more subregions end up as some kind of complex multi-step or multi-stage ko. For simplicity, in this paper we concentrate only on eyes for which the defender does not need to fight ko. Our solver correctly deals with double ko etc. as long as all ko are in the same region.

4.2 The Dynamic Decomposition Search Algorithm

Let R be the region at the root position, and R_w be the *working region* that the algorithm is currently searching in. At

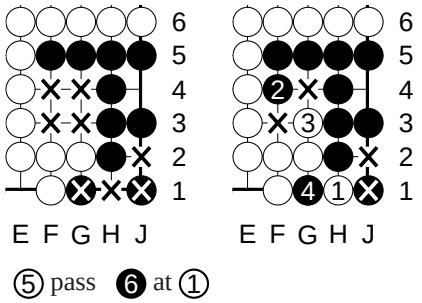

Figure 3: Interacting regions in ko with the divide and conquer approach.

the start, set $R_w = R$. Then *dynamic decomposition search* (DDS) works as follows:

1. If there is an eye in R_w, the defender wins.
2. Otherwise, if no eye space remains in R_w, the attacker wins in R_w.
3. Before generating moves, the region R_w is tested for a possible split into subpositions. Both safe attacker stones and crucial defender stones are used for splitting. This check is done at every newly encountered nonterminal position.
4. Suppose that a position is already partitioned into several subpositions $R_1 \cdots R_k$. If the defender is to play and a move in R_i is chosen by the search control (see the next subsection), then the working region R_w is restricted to R_i. Below this position, moves are generated only in R_i, or an even smaller region when further decompositions occur. This reduces the number of possible moves and the search depth to reach terminal positions. If the defender finds an eye in one of the subregions $R_1 \cdots R_n$, the defender wins. If no eye is found in any subregion, the attacker wins.
5. If the attacker is to play, all moves in R_w are tried.

4.3 Search Control

If there are several subregions, the defender must select one to expand the search in. Df-pn with DDS selects the move to play based on proof and disproof numbers. This dynamically selects a most promising working region at each step. Figure 4 shows an example. Let the defender be a player to prove a position, and the numbers on the board be proof numbers. In this figure, Black plays at **B6**, because it has the smallest proof number. The working region is narrowed to the left side. White answers only in the left subregion after Black's **B6**. Assume that the proof number at **B6** is changed to 5 after exploring positions below Black's play at **B6**. Then, Black plays at **G6** because it becomes the smallest proof number. The working region switches to the right subregion.

4.4 Using the Transposition Table in DDS

In a normal transposition table, Zobrist hashing [12] maps a full board position to its hash key. However, in DDS we must distinguish between different working regions. For example, if a position contains two subregions A and B, there can be three cases for move generation: only in A, only in B, and in both A and B. In order to differentiate these cases, we encode the working region into the hash key as well.

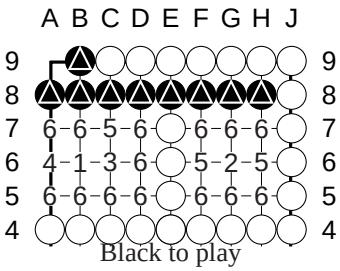

Figure 4: Example of using proof numbers to select the working region.

5 Experimental Results

5.1 Setup of Experiments

Even though there is a large amount of literature on tsume-Go, there are almost no specialized collections of one-eye problems. Landman [4] has a collection of small examples. We created our own test collection with more challenging instances, available at `http://www.cs.ualberta.ca/~games/go/oneeye/`. Each test problem can be solved for either color moving first. Some problems are only interesting if one particular player goes first, and are very easy for the other.

We used two test suites in the experiments. The first test suite, the *toy problem collection* (TOY), contains 13 test positions (26 problems) that are already completely or mostly split into independent problems at the root. Figure 5 illustrates an example of this kind of problem. These problems were used mainly to verify that the decomposition approach works. The second test suite is the *standard problem collection* (STANDARD), an extension of the test collection used in [3]. It contains 81 test positions (162 problems). Some problems are hard for our previous one-eye solver. Figure 6 shows a representative example.

We compare two versions of our solver, with and without dynamic decomposition search (DDS). The version without DDS, no-DDS, is based on the solver described in [3]. It

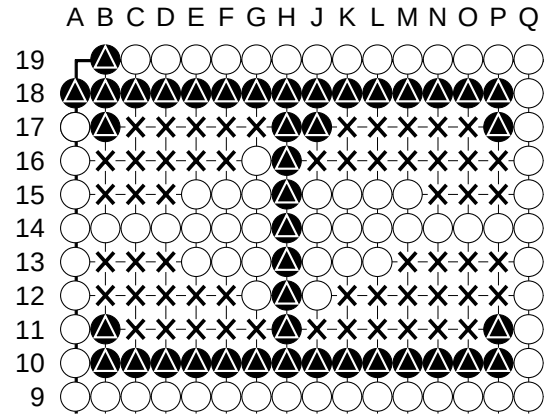

Figure 5: Example of a one-eye problem that is already split (divide-conquer.12.sgf, Black to live by playing at **O12**).

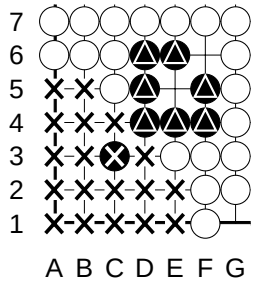

Figure 6: Example of a hard problem (oneeyee.1.sgf, Black lives with **B2**).

is improved by adding heuristic initialization of proof and disproof numbers at leaf nodes. All experiments were performed on an Athlon XP 2800+ with a 300 MB transposition table. The time limit was 5 minutes per problem.

5.2 Results

Tables 1 and 2 compare the solving abilities of DDS and no-DDS. More positions are solved by using DDS in TOY. Moreover, all problems solved by no-DDS were also solved by DDS. On the other hand, both versions solved the same subset of problems in STANDARD. The improvement achieved by DDS is a factor of 66 in TOY and 1.2 in STANDARD in total execution time. This indicates that DDS surpasses the abilities of our previous df-pn solver.

On average, DDS is about 13% slower in terms of node expansions per second (see Table 2). However, sometimes

Table 1: Performance comparison for DDS and no-DDS in TOY. All statistics are computed for 23 problems solved by both program versions.

	Number of problems solved	Total time (sec)	Total nodes expanded	Nodes expanded per second
No-DDS	23	52	2,169,239	41,636
DDS	26	0.79	49,433	62,573
Total Problems	26	-	-	-

Table 2: Performance comparison for DDS and no-DDS in STANDARD. All statistics are computed for 157 problems solved by both program versions.

	Number of problems solved	Total time (sec)	Total nodes expanded	Nodes expanded per second
No-DDS	157	1,975	84,084,752	42,573
DDS	157	1,645	62,129,738	37,774
Total Problems	162	-	-	-

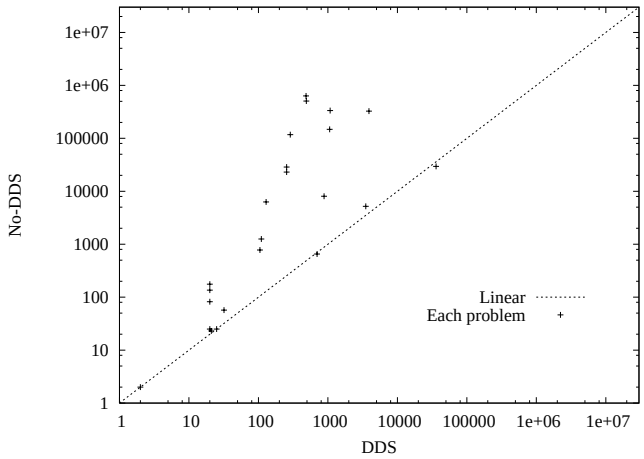

Figure 7: Node expansions for toy problems solved by both versions.

DDS is faster, especially in TOY (see Table 1). In particular, with decompositions at or near the root, DDS can concentrate on a smaller region, which speeds up basic operations such as detecting potential eye points and move generation.

Figure 7 compares node expansions of both solvers for each problem in TOY. The number of nodes explored by DDS is plotted on the X-axis against no-DDS on the Y-axis on logarithmic scales. In points above the diagonal DDS performed better. Except for one problem DDS expanded at most as many nodes as no-DDS, and often dramatically less. The performance of DDS scales exponentially better in the size of problems. For example, DDS solved the position in Figure 5 in 1,075 nodes, while no-DDS needed 334,718 nodes. This is not surprising, since all positions in this set are ideal for DDS, while no-DDS suffers from combinational explosion.

Figures 8 and 9 present the results for STANDARD. None of these problems were designed with decomposition in mind. In contrast to Figure 7, there are more problems where DDS was slower. However, on average DDS explores less nodes and needs less execution time. This is especially true for the larger problems, so DDS seems to scale better. DDS sometimes improves the performance by a large margin. For example, DDS needed 360,163 nodes in 7.5 seconds for the position in Figure 6, whereas no-DDS explored 1,732,845 nodes in 35.6 seconds. In this position, decompositions triggered by black crucial stones and white safe stones reaching the borders of the board seem to occur frequently.

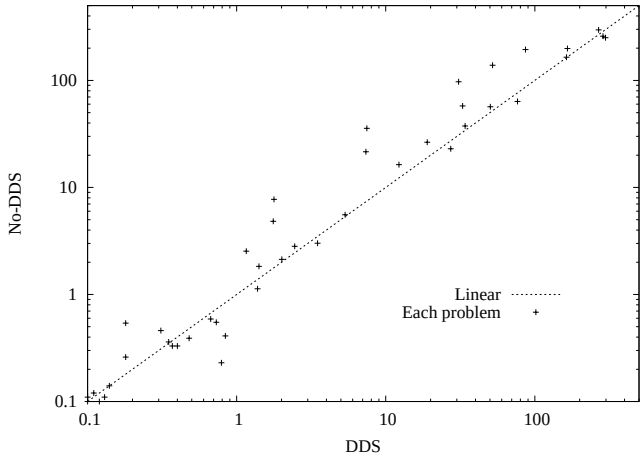

Figure 8: Execution time for standard problems solved by both versions.

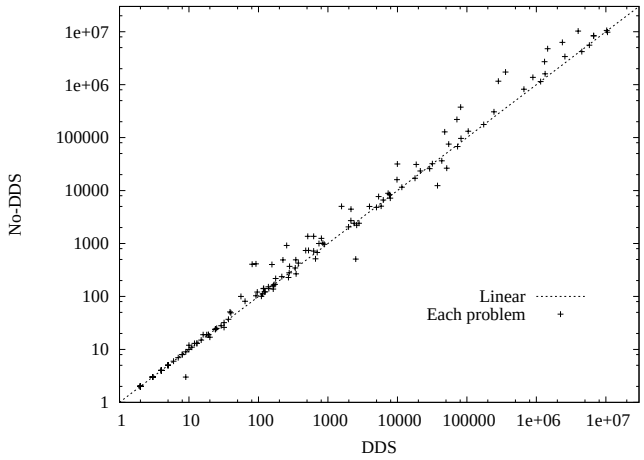

Figure 9: Node expansion for standard problems solved by both versions.

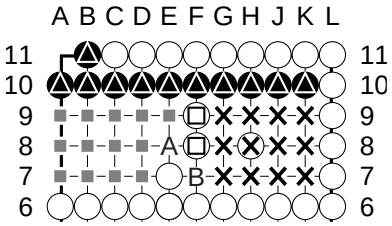

Figure 10: Relaxed decomposition.

In the hard problems of this set, the percentage of nodes in which decompositions are possible varies from 16% to 50%. In Figure 6, DDS detected 127,479 (35.3%) decompositions.

6 A Relaxed Decomposition Model

DDS is limited in the way that splits are recognized. The only points used to split positions are those occupied by safe attacker and crucial defender stones. In this section we introduce a less rigid decomposition that uses "almost safe" attacker stones as well. Figure 10 shows an example. If we assume that the two white stones marked by squares are safe, then we can split the area into two subregions, a left subregion marked by small grey squares and a right region marked by crosses. The attacker can always make the two marked white stones safe by following a simple *miai* connection strategy: Whenever the defender plays either *connection point* A or B, reply on the other point. In our *relaxed decomposition* model, we use such stones for splitting a position. However, during the search we must handle the case where a miai connection is attacked by the defender.

In this case, our approach extends the search to the union of the affected subregions. We explain the algorithm in detail with the help of Figure 10. In this figure, let region R_1 consist of the empty point A and all points marked by filled squares. Region R_2 contains B and all points marked by crosses. The connection points A and B together form a miai connection from safe attacker stones to an almost safe attacker block. Without loss of generality, assume that the defender starts playing in R_1. Then *Relaxed Decomposition Search* (RDS) defines the strategies of both players as follows:

1. If both A and B are empty, both players are restricted to play moves in R_1.

2. If either A or B contains a stone of the attacker, the relaxed decomposition has changed into a normal decomposition. Both players keep playing in R_1.

3. Otherwise, if at least one of A and B contains a stone of the defender, and the other point is empty or also occupied by the defender, the region is extended and both players continue playing in $R_1 \cup R_2$.

In RDS, as long as the defender does not play at A, both players stick to play in R_1. However, if the defender plays at A and the attacker does not respond at B, then the defender can invade R_2. As in DDS, the defender can switch

between trying moves in R_1 and R_2 at the root. This process is controlled by proof numbers.

The following lemma is needed to prove correctness of RDS.

Lemma 6.1 *Assume the following:*

1. *Position P contains region R which is split into two subregions R_1 and R_2 by relaxed decomposition.*
2. *The attacker is to play in P.*
3. *Two connection points A in R_1 and B in R_2 are empty.*
4. *If the attacker plays at A in P, the defender can still create an eye unconditionally (without ko) in R_1.*

Then, the defender can create an eye in R_1 unconditionally for the position after the attacker plays a move in R_2 for P.

We give a proof sketch for the case of a DAG. Let P_1 be the node after the attacker plays a move at A for P, and P_2 be the node after the attacker plays a move in R_2 for P. We prove that the defender can make an eye for P_2 by following the winning strategy for P_1. The lemma is proven by induction on the maximum depth d of a terminal node in the proof graph of P_1.

Case 0: $d = 0$ If P_1 is a terminal position, the same eye exists in both P_1 and P_2.

Case 1: $d = 1$ The defender can create an eye by making move $m \neq A$ in R_1 for P_1. Since P_1 is identical to P_2 within $R_1 - \{A\}$, m is legal in P_2 and also creates an eye there.

Case 2: induction step Assume that Lemma 6.1 holds for all $d \leq k$. We prove that the lemma also holds for $d = k + 2$. Let $m \in R_1 - \{A\}$ be the winning defender move in P_1, Q_1 be P_1's child after playing m for P_1, and $n_1, n_2, \cdots, n_l$ be Q_1's children. Let n_1 be the node after the attacker passes for Q_1. Since $m \in R_1 - \{A\}$, m is legal for P_2. Let Q_2 be P_2's child by playing m, o_1 be Q_2's child after the attacker plays at A, $o_2 \cdots o_p$ be Q_2's children after moves in R_2, and $o_{p+1} \cdots o_{p+q}$ be Q_2's children from moves in R_1. The defender can make an eye for $o_{p+1} \cdots o_{p+q}$ by the assumption of Lemma 6.1. R_1 is completely separated from R_2 in o_1 and n_1. Moreover, since o_1 and n_1 are identical positions in R_1, the defender can create an eye in o_1. By induction, since o_1 has depth $d \leq k$, the defender can create an eye for $o_2 \cdots o_p$. Thus, the lemma is proven for the case of $k + 2$.

The following theorem guarantees the correctness of RDS in the case that an eye is found.

Theorem 6.1 *Assume that R is split into two subregions R_1 and R_2 by RDS. If the result of RDS shows that the defender can create an eye unconditionally (without ko) in either R_1 or R_2, then that eye can always be made against any attacker strategy in $R_1 \cup R_2$.*

In the following, we call the proof graph created by RDS the *RDS proof graph*, and a proof graph for the whole region $R_1 \cup R_2$ an *original proof graph*.

We present a proof sketch which shows that each RDS proof graph can be converted into an original proof graph, for the case where the RDS proof graph is a DAG. We use induction on depth of a terminal node in the RDS proof graph. We only explain the first case of RDS with the help of Figure 10. The other two cases are trivial, because searching either with completely separated subregions or with the whole region is performed in those cases.

Assume without loss of generality that the first defender move is in R_1. As above, if the RDS graph contains only a terminal node, an eye already exists and the RDS proof graph also works as an original proof graph.

Otherwise, let n be the root of the RDS proof graph. Assume that by induction n's descendants in the RDS graph have been converted to original proof graphs.

- If n is an OR node, n's move m leading to n's child n_c in the RDS proof graph is also legal for searching in $R_1 \cup R_2$. n_c's RDS proof graph can be converted to n_c's original proof graph by the induction assumption. Hence, we can construct n's original proof graph by adding a branch m from n to n_c's original proof graph.

- If n is an AND node, assume that n's children $n_{c_1}, \cdots, n_{c_k}$ in R_1 have proof graphs. Let n_{c_1} be n's child after the attacker plays a move at A, $n_{c_{k+1}} \cdots n_{c_l}$ be n's children by playing in R_2. We need to prove that $n_{c_{k+1}} \cdots n_{c_l}$ have original proof graphs. n_{c_1} guarantees that an eye can be made in R_1, since R_1 is completely separated from R_2. By applying Lemma 6.1 to $n_{c_{k+1}} \cdots n_{c_l}$ based on n_{c_1}'s proof graph, the defender can make an eye for $n_{c_{k+1}} \cdots n_{c_l}$. Hence, $n_{c_{k+1}} \cdots n_{c_l}$ have original proof graphs.

We believe that our lemma and theorem also hold for cyclic graphs in the case where the eye can be made unconditionally. However, we need a different approach to prove our conjecture for cyclic graphs, because we use a property of DAGs in proving the theorem by induction. The induction in our proof uses the fact that children have a depth that is at least 1 smaller than their parents. This property does not hold for cycles.

RDS can split positions more frequently than DDS. The approach can be generalized to more than two relaxed split subregions, as long as all the miai connections to safe attacker stones are disjoint. However, the completeness of the relaxed decomposition algorithm is not known yet. To prove that no eye is possible, the worst case scenario might require re-searches in the whole region. In this case, we must devise an efficient way for re-searches.

7 Conclusions and Future Work

In this paper, we presented a method that dynamically decomposes a position into sub-positions during search. The

results of this dynamic decomposition search are encouraging. In many problems, DDS is able to reduce the search space by a large margin, thereby enabling the one-eye solver to solve harder problems more quickly. However, there are still a lot of unexplored topics such as:

- The current version of DDS is limited to dealing with ko fights only in the same region. To overcome this problem, we need to find a detailed ko status and ko threat status of each divided region, and combine them.
- Investigating relaxed decomposition search is a challenging topic from both the theoretical and practical point of view. Furthermore, splitting a position in a more aggressive way such as by using divider patterns [6] is an interesting extension of this research topic.
- Applying the ideas to other parts of Go, such as connections, territories, and tsume-Go, or other games such as Hex will be a challenging topic. In particular, tsume-Go is an interesting domain for further investigations. In tsume-Go, the decomposition will be more complicated. Suppose that a region is split into two completely separated rooms A and B. There are several possibilities to be considered, such as making (1) two eyes at A, (2) one eye at A and the other eye at B, (3) two eyes at B, or (4) one eye either at A or at B if one eye already exists. Local searches must distinguish between *sente* and *gote*. For example, the position in Figure 11 has two subregions, a left subregion R_1 marked by small grey squares and a right subregion R_2 marked by crosses. Move **B6** in R_1 makes one and a half eye and move **D6** in R_2 makes half an eye. Black can live with **B6**, since White cannot play both **B4** and **D6**. To solve such a problem with two separate searches in R_1 and R_2, return values such as "1.5 eyes" or "0.5 eyes" [4] must be recognized by the search.
- Incorporating DDS into a complete Go-playing program is an important topic. We will probably have to heuristically decompose positions, since many positions in the real games are not closed off by safe attacker stones.

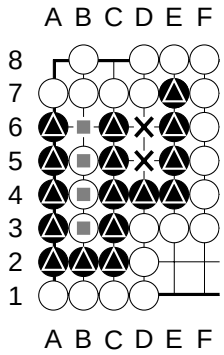

Figure 11: Decomposition for tsume-Go.

Acknowledgments

We would like to thank Adi Botea for reading drafts of the paper and giving valuable comments. Financial support was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Alberta Informatics Circle of Research Excellence (iCORE).

Bibliography

[1] E. Berlekamp, J. Conway, and R. Guy. *Winning Ways*. Academic Press, 1982.

[2] M. Campbell, A. Joseph Hoane Jr., and F.-h. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[3] A. Kishimoto and M. Müller. Df-pn in Go: Application to the one-eye problem. In *Advances in Computer Games. Many Games, Many Challenges*, pages 125–141. Kluwer Academic Publishers, 2003.

[4] H. Landman. Eyespace values in Go. In R. Nowakowski, editor, *Games of No Chance*, pages 227–257. Cambridge University Press, 1996.

[5] M. Müller. Decomposition search: A combinatorial games approach to game tree search, with applications to solving go endgames. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI'99)*, volume 1, pages 578–583, 1999.

[6] M. Müller. Computer Go. *Artificial Intelligence*, 134:145–179, 2002.

[7] Ayumu Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications*. PhD thesis, Department of Information Science, University of Tokyo, 2002.

[8] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, 1997.

[9] R. Vilà and T. Cazenave. When one eye is sufficient: A static approach classification. In *Advances in Computer Games. Many Games, Many Challenges*, pages 109–124, 2003.

[10] T. Wolf. The program GoTools and its computer-generated tsume Go database. In Hitoshi Matsubara, editor, *Game Programming Workshop (GPW)*, pages 84–96. Computer Shogi Association, 1994.

[11] T. Wolf. Forwarded pruning and other heuristic search techniques in tsume go. *Information Sciences*, 122(1):59–76, 2000.

[12] A. L. Zobrist. A new hashing method with applications for game playing. Technical report, Department of Computer Science, University of Wisconsin, Madison, 1970.