Preprint, *Computers and Games 2000*, LNCS (Springer)

Lambda-search in game trees – with application to Go

Thomas Thomsen

Stockholmsgade 11, 4. th.
DK-2100 Copenhagen, Denmark
thomas@t-t.dk

Abstract. This paper proposes a new method for searching two-valued (binary) game trees in games like chess or Go. Lambda-search uses null-moves together with different orders of threat-sequences (so-called lambda-trees), focusing the search on threats and threat-aversions, but still guaranteeing to find the minimax value (provided that the game-rules allow passing or zugzwang is not a motive). Using negligible working memory in itself, the method seems able to offer a large relative reduction in search space over standard alpha-beta comparable to the relative reduction in search space of alpha-beta over minimax, among other things depending upon how non-uniform the search tree is. Lambda-search is compared to other resembling approaches, such as null-move pruning and proof-number search, and it is explained how the concept and context of different orders of lambda-trees may ease and inspire the implementation of abstract game-specific knowledge. This is illustrated on open-space Go block tactics, distinguishing between different orders of ladders, and offering some possible grounding work regarding an abstract formalization of the concept of relevancy-zones (zones outside of which added stones of any colour cannot change the status of the given problem).

Keywords: binary tree search, threat-sequences, null-moves, proof-number search, abstract game-knowledge, Go block tactics

1 Introduction

λ-search is a general search method for two-valued (binary) goal-search, being inspired by the so-called null-move pruning heuristic known from computer chess, but used in a different and much more well-defined way, operating with direct threats, threats of forced threat-sequences, and so on.

To give a preliminary idea of what λ-search is about, consider the goal of mating in chess. A direct threat on the king is called a *check*, and we call a sequence of checks that ends up mating the king a forced mating check-sequence. However, a *threat* of such a forced mating check-sequence is not necessarily a check-move itself, and such

a meta-threat – being of a second order compared to the first-order check moves in the check-sequence – can be a more quiet, enclosing move.[1]

Similarly, in Go, a direct threat on a block of connected stones is called an *atari*. A forced atari-sequence resulting in the capture of the block is called a working *ladder* (Japanese: *shicho*). Hence, in this context, a (meta-)threat of a forced threat-sequence would be a ladder-threat, which does not have to be an atari itself. Consider figure 1: Black cannot capture the white stone *a* in a ladder, because of the ladder-block (the other white stone). An example of such a failed attempt is shown in figure 2. Now, providing that black gets a free move (a move that is answered with a pass by white), there are a number of ways in which black can make the ladder work. One of the possibilities could be playing a free move at *c* in figure 3. This threatens the ladder shown in figure 4. Another possibility could be playing at *d* in figure 3, threatening the ladder shown in figure 5.

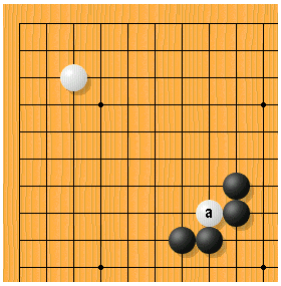

Fig. 1. How to capture a?

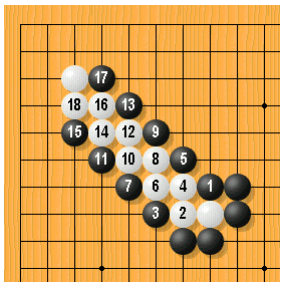

Fig. 2. A failing ladder (λ^1=0)

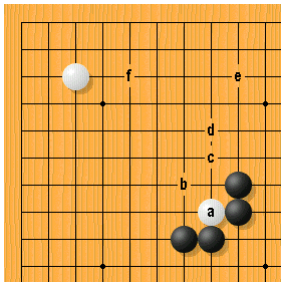

Fig. 3. Black λ^2-moves?

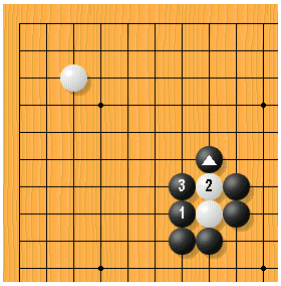

Fig. 4. Working ladder (λ^1=1) after black *c*

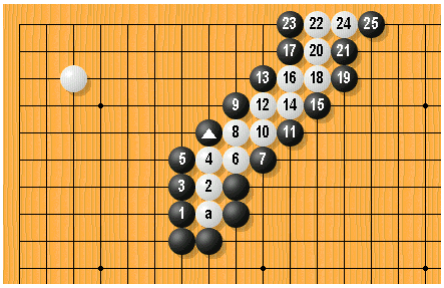

Fig. 5. Working ladder (λ^1=1) after black *d*

The moves *c* and *d* in figure 3 are thus more indirect threats on the white stone *a*, compared to a direct (first-order) atari, and this possible generalization of the concept of threats is the main theme of the present paper.

---

[1] Note: for readers not familiar with Go, the appendix contains a chess-example similar to the Go-example given below. In chess, goals other than mating could be, for instance, trapping the opponent's queen, creating a passed pawn, or promoting a pawn.

To introduce the terminology used in this paper, we denote a direct threat-sequence a λ^1-tree, consisting solely of λ^1-moves (checks/ataris for the attacker and moves averting these for the defender). The λ^1-tree is a standard search tree solvable by means of any minimax search technique, with value 1 (success for the attacker) or 0 (success for the defender).

At the next level of abstraction, a λ^2-move for the attacker is a move that threatens a λ^1-tree with value 1, if the attacker is allowed to move again. Figure 6 depicts all possible black λ^2-moves given unlimited search depth; i.e., all black moves threatening a working ladder (or a direct capture; hence the two atari-moves on white *a* are also λ^2-moves) if left unanswered. If e.g. 1 in figure 7 (= *c* in figure 3) is chosen, black threatens the ladder depicted in figure 4. Figure 7 then shows all possible white moves averting that threat. In the general λ^2-tree, these two white moves in Figure 7 are hence λ^2-children of black 1, and in figure 8, the two black moves are λ^2-children of white 2 (again reviving the ladder-threat).

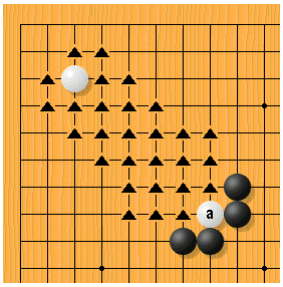

Fig. 6. Black λ^2-moves to kill *a*

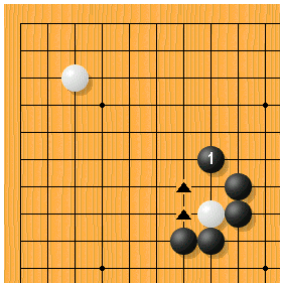

Fig. 7. White λ^2-moves after black 1

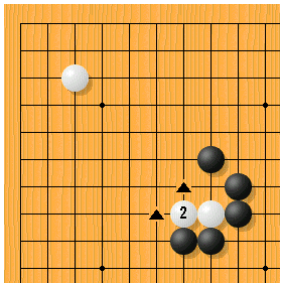

Fig. 8. Black λ^2-moves after white 2

In this way, a λ^2-tree can be built out of λ^2-moves, these being threats of threat-sequences (ladder-threats) for the attacker, and aversions of threat-sequences (ladder-breakers) for the defender.[2] See the appendix, figures A1-A3, for a similar chess-example.

In the context of Go block tactics, a λ^2-tree can be interpreted as a net or loose ladder (Japanese: *geta* or *yurumi shicho*), but the order of λ can easily be > 2. For instance, loose loose ladders (λ^3-trees) are quite common in open-space Go block tactics, the λ^3-trees being built up of λ^3-moves, originating from λ^2-trees. Tsume-go problems (enclosed life/death problems) or semeai (race-to-capture) usually involves high-order λ^n-trees (mostly $n \geq 5$), and connection problems (trying to connect two blocks of stones) often features high λ-orders, too. For example, in figure 9, white *a* can be killed in a λ^7-tree (the solution is black *b*, 21 plies/half-moves), whereas in fig-

[2] As most readers with a basic knowledge of Go would know, the best black λ^2-move is *b* in figure 3, after which there are no white λ^2-moves. Another way of stating this is that after black *b*, there is no way for white to avoid being killed in a subsequent ladder (λ^1-tree), and so white can be declared unconditionally dead.

ure 10, black *a* and *b* can be connected into one block in a λ^3-tree (the solution is black *c* or *d*, 15 plies).[3]

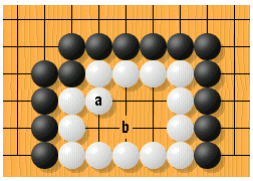

Fig. 9. Black to kill white *a* (λ^7)

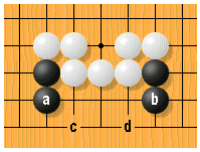

Fig. 10. Black to connect *a* and *b* (λ^3)

As it is shown in section 3, compared to standard alpha-beta, λ-search can often yield large reductions in the number of positions generated in order to solve a given problem, especially if the search tree is highly non-uniform in a way exploitable by λ-search. Hence, the λ-search methodology could probably prove useful for goal-search in almost any two-player, full-information, deterministic zero-sum game.[4]

The point about λ-search is that in each λ-tree, all legal moves are not just being blindly generated and ploughed through, but instead the search focuses on those making and averting (more or less direct) threats. This implicit move-ordering directs the search towards its goal, and even though the λ-search methodology does not in itself contain any knowledge of the concrete game-problem analyzed, λ-moves often seem to have at least some minimum of meaning or purpose (cf. figures 6-8, or figures A1-A3 in the appendix), in contrast to the often random-looking move sequences generated by no-knowledge alpha-beta search.

In addition to this implicit threat-based move-ordering, the λ-search approach can often ease and inspire the implementation of abstract game-specific knowledge. For instance, regarding λ^1-moves in chess mating problems, there are only three ways of averting a check (capturing the threatening piece, putting a piece in between, or moving the king). Similarly, regarding λ^1-moves in Go block tactics (ladders), the attacker must always play on one of the defender's liberties, whereas the defender must always play on his remaining liberty, or capture a surrounding block. Section 4 describes an attempt at formulating some abstract/topological Go-tactical knowledge regarding general λ^n-trees, these abstract rules relying both upon the value of the λ-order *n*, and on information generated during search of lower-order λ-trees.

---

[3] Note generally that if $\lambda^n = 1$, the attacker has to spend $n + 1$ extra moves (moves that can be answered by a pass/tenuki by the defender) in order to "execute" the goal. Hence, in figure 9, white *a* can be conceived of as having $n + 1 = 8$ effective liberties, instead of only the 5 real/visible ones. If the black block surrounding white was vulnerable, such knowledge would be crucial in for instance semeai problems (cf. [10]), or regarding the difficult problem of "opening up" tsume-go problems (cf. [12]).

[4] Provided that the game-rules allow passing, or zugzwang (move-compulsion) is not always a motive for obtaining goals.

2 Formalizing the λ-search Method

In this section, the λ-search method is formalized in its most general form, not tied to the context of any specific game. The λ^n-tree is defined as follows:

Definition 1: A λ^n-tree is a search tree for trying to achieve a single well-defined goal, the search tree consisting solely of λ^n-moves (defined in definition 2), and a λ_a^n-tree being a λ^n-tree where the attacker moves first. The minimax value of a λ^n-tree is either 1 (success for the attacker) or 0 (success for the defender).[5] A node is a leaf (terminal node) if the node has no children because there are no legal λ^n-moves following it. If this is so, the value of the leaf is 1 if the defender is to move, and 0 if the attacker is to move.

The λ^0-tree is particularly simple: $\lambda_a^0 = 1$ if the attacker's goal can be obtained directly by at least one of the attacker's legal moves, and $\lambda_a^0 = 0$ otherwise.

Definition 2: A λ^n-move is a move with the following characteristics. If the *attacker* is to move, it is a move that implies – if the defender passes – that there exists at least one subsequent λ_a^i-tree with value 1, $0 \le i \le n-1$. If the *defender* is to move, it is a move that implies that there does not exist any subsequent λ_a^i-tree with value 1, $0 \le i \le n-1$.

Example (Go block tactics): Regarding the construction of a λ^2-tree, cf. figure 3. Here, a black play at *c* is a λ^2-move. After *c*, provided that white passes, $\lambda_a^0 = 0$ (the block cannot be captured directly), but $\lambda_a^1 = 1$ (the block can be captured in a ladder, cf. figure 4).

After black plays this λ^2-move at *c*, we have the situation in figure 7. Here, a white move at any of the two marked points is a λ^2-child of the black λ^2-move, since after any of these two white moves, $\lambda_a^0 = \lambda_a^1 = 0$ (black cannot follow up with neither a direct capture nor a working ladder).

The λ-method can be illustrated as in figure 11. In this figure, black *a* is a λ^n-move, because – if white passes – it can be followed up with a λ_a^{n-1}-tree with value 1, cf. the small tree to the left. Considering the white move *b*, this is likewise a λ^n-move, since it is not followed by a λ_a^{n-1}-tree with value 1, cf. the small tree to the right.[6] Other legal black and white moves are pruned off, and hence the λ^n-tree generally has a much smaller average branching factor than the full search tree. Note that in all the three depicted λ-trees the attacker moves first, and note also that the algorithm works recursively in the "horizontal" direction, so that the moves in the small λ^{n-1}-trees to the left and right are in turn generated by λ^{n-2}-trees, and so on downwards, ultimatively ending up with λ^0-trees. Note finally that a λ^{n-1}-tree generally contains much fewer nodes

---

[5] The terms "attacker" and "defender" are arbitrary and could just as well be "left" and "right". For instance, in Go, "attacking" could just as well be trying to connect two blocks of stones, and "defending" trying to keep them disconnected.

[6] In fact, small λ_a^{n-2}-trees, λ_a^{n-3}-trees, and so on, should also be included at the left and right (cf. definition 2), but these are omitted here for the sake of clarity.

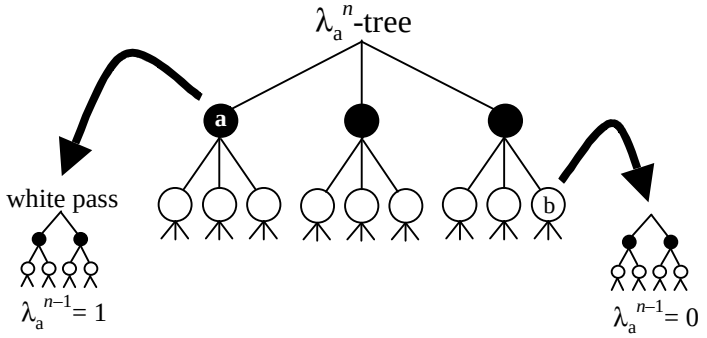

Fig. 11. The λ^n-tree

than a λ^n-tree. Having made the above definitions, it is time to introduce an important theorem:

Theorem 1 (confidence): If a λ_a^n-tree search returns the value 1 (success for the attacker), this is the minimax value of the position – the attacker's goal can be achieved with absolute confidence.

Proof: For a λ_a^n-tree to return the value 1, the attacker must in each variation be able to force the defender into situations where he has no λ^n-moves (cf. definition 1). The defender not having any λ^n-move means that no matter how the defender plays, the attacker can follow up with a lower-order λ-tree with value 1: $\lambda_a^i = 1$, $0 \leq i \leq n - 1$. By the same reasoning as before this in turn means that the attacker can force the defender into situations where he has no λ^i-moves. By recursion, this argument can be repeated for lower and lower orders of λ-trees, until we (in at most n steps) end up with a λ^0-tree. And if a λ_a^0-tree has value 1, the goal is obtained directly (cf. definition 1). □

This theorem is crucial for establishing the *reliability* of λ-search. As theorem 1 shows, if a λ_a^n-tree returns the value 1, the attacker's success *is* a proven success no matter how the defender tries to refute it, so if a λ-search finds a win for the attacker, one can have absolute confidence in it.

Theorem 1 is obviously not symmetric. If a λ_a^n-tree search returns the value 0, this does not entail that there could not be a forced attacker's win in the full game-tree. It only means that the defender cannot lose by means of attacker's λ^n-moves, but there might very well be a forced win for the attacker lying in wait in trees of order n+1 or higher.

Example (Go block tactics): If black plays a working ladder (λ^1=1), he is certain to kill the white block: the defender cannot avoid being killed. But the other way round,

if a white block cannot be killed in a ladder, this does not, of course, imply that it could not, for instance, be killed in a loose ladder.

Seen in the light of the above, given some specific goal for the attacker to achieve, and provided that the attacker cannot achieve the goal directly within the first move ($\lambda_a^0 = 0$), the strategy will be to first try out a λ_a^1-tree. If $\lambda_a^1 = 1$, we are finished, cf. theorem 1. If $\lambda_a^1 = 0$, we continue with a λ_a^2-search, and so on until time runs out. If we end up with $\lambda_a^n = 0$ for some value of *n*, we cannot say anything authoritative on the status of the problem, other than that the defender survives a λ_a^n-attack.

The next question is what happens asymptotically as λ^n-trees of higher and higher order get searched? Provided that we know a priori that the attacker's goal *can* be reached within at most *d* plies by optimal play (*d* being an odd number), can we be sure that the value of λ_a^n will converge to 1 for some *n*, as *n* tends towards infinity?

If the game-rules allow passing, the answer is yes, and in that case it is possible to prove that $n \leq (d–1)/2$, as shown in theorem 2 below. First a lemma:

Lemma 1. Consider a full search tree consisting of all legal moves for both players. In this search tree, consider a node *x* where the attacker is on the move. Assume that we are not considering the root node (move #1), but some attacker's move deeper down the tree (i.e., some odd move number ≥ 3). Consider the sub-tree unfolding from node *x*. For the attacker to be able to force the defender into a λ_a^n-tree with value 1 expanding from node *x*, the attacker must have been *threatening* $\lambda_a^n = 1$ when he was on the move last time. Had he not been threatening $\lambda_a^n = 1$, the defender could have passed, after which (cf. definition 2) the value of a subsequent λ_a^n-tree would have been 0.

Example (Go block tactics): in order to be able to force the defender into a working ladder ($\lambda_a^1 = 1$), the preceding attacker's move must have been a ladder-threat. If this had not been so, the defender could have passed, and the ladder would still not work.

Theorem 2 (convergence). Consider again the full search tree consisting of all legal moves for both players. Assume that we know a priori that there exists a forced win for the attacker in at most *d* plies, *d* being some odd number ≥ 1. Consider a node *y* at depth $d – 1$ (with the attacker to move) after which the forced win is executed at depth *d*. This means that the tree expanding from *y* is a λ_a^0-tree with value 1. Using Lemma 1, this again means that the attacker must have been *threatening* $\lambda_a^0 = 1$ when he played last time (i.e., played a move at depth $d – 2$). Threatening $\lambda_a^0 = 1$ is the same as playing a λ^1-move; hence the attacker must have played a λ^1-move at depth $d – 2$.

In order for the attacker to be able to force the defender into this λ^1-move at depth $d – 2$, it must have been part of a λ_a^1-tree with value 1. The first attacker's move in this tree can be at any depth d_1, where $1 \leq d_1 \leq d – 2$. If $d_1 = 1$, we are finished (hence establishing $n = 1$). Else, if $3 \leq d_1 \leq d – 2$, in order to force the defender into this λ_a^1-tree with value 1, the attacker must have been threatening $\lambda_a^1 = 1$ (= playing a λ^2-move) at some depth d_2, where $1 \leq d_2 \leq d_1 – 2$.

This goes on recursively for λ-trees of higher and higher order (each time moving at least two plies upwards in the full search tree), so in the worst case we end up with a tree of order $n = (d–1)/2$. This reasoning can be repeated for all nodes *y* at depth $d –$

1 after which the forced win is executed at depth *d*, and hence it is proved that $n \leq (d-1)/2$. □

Example (Go block tactics): Assume that we know a priori that the attacker can kill the defender in at most 5 plies. The attacker's move #5 must have been the capture (taking the stones off the board), so the attacker's move #3 must have been an atari (λ^1-move) – otherwise the defender could have passed, and the stones could not have been taken off the board at move #5. Regarding the attacker's move #1, it can have been a(nother) λ^1-move if the whole killing sequence is a working ladder (λ^1-tree with value 1). Otherwise, the first attacker's move could have been a λ^2-move (loose ladder move), threatening $\lambda_a^1 = 1$ (an ordinary ladder). But if the attacker's move #1 had not been either a λ^1-move or a λ^2-move, the defender could have passed at move #2, and there would have been no way for the attacker to establish a working ladder (λ^1-tree) at move #3, and hence no way to kill the defender at move #5. Thus, if $d = 5$, the λ-search can only be of order 1 or 2, meaning that if $d = 5$ and a λ^2-search returns value 0, the problem cannot be solved in 5 plies. This corresponds to the inequality, since the formula states that $n \leq (d-1)/2 = 2$.

Considering games like chess where passing is not allowed, theorem 2 does not hold, since the ability to pass is a prerequisite for Lemma 1. However, apart from zugzwang-motives (e.g. in late chess endgames), this is not a problem if the defender has always at least some harmless/non-suicidal move to play instead of a pass move. Hence, for practical purposes, theorem 2 also applies to goals such as e.g. mating in middle-game chess positions. And it should be noted that theorem 1 applies whether passing is allowed or not.

Thus, theorems 1 and 2 show that λ-search is not to be interpreted as some heuristic forward-pruning technique, since a λ-search – in the worst case searched to λ-order $n = (d-1)/2$ – returns the minimax value, provided that passing is allowed or zugzwang is not a motive. Some pseudo-code containing the λ-search methodology can be found in [11].

3 Comparing λ-search with other Search Methods

Having offered the proofs of confidence and convergence, the question is how effective the λ-search method really is compared to other techniques, such as, e.g., standard alpha-beta, alpha-beta with null-move-pruning, or proof-number search?

3.1 Standard alpha-beta

Given some simplifying assumptions it is possible to indicate the effectiveness of λ-search compared to standard alpha-beta. Below, it is shown that this effectiveness generally depends on the branching factor of the λ-search trees compared to the branching factor of the full search tree, combined with the λ-order *n*.

Consider a general λ-search tree of some order *n*, with the attacker moving first, and assume that an attacker's win exists in at most *d* plies (*d* being uneven). Also as-

sume that passing is allowed or zugzwang is not a motive. We will now try to count the total number of positions generated in order to search such a λ^n-tree, including the positions generated in the lower-order λ-trees. It is assumed that minimax is used in both the λ-tree, and as the standard reference algorithm, for reasons to be explained.

Assume furthermore that *B* is the average branching factor of the full search tree (= the number of legal moves in each position: in chess, around 30-40, and in Go around 200-250). Assume furthermore that all λ-trees have the same average branching factor, *b* (*b* being ≤ *B*). In order to generate the attacker's moves at depth 1 in the λ^n-tree, all legal attacker's moves must be tried out, to see whether or not they can be followed by some λ^i-tree ($i \leq n-1$) with value 1 (cf. *a* in figure 11). This amounts to Bn λ^i-tree searches (the λ^i-trees being of order $i = 0, 1, \ldots, n-1$). And at depth 2 in the λ^n-tree, for each of the defender's λ^n-nodes, all legal defender's moves must be tried out, to see whether or not they are followed by λ^i-trees ($i \leq n-1$) with value 0 (cf. *b* in figure 11). This amounts to a total of bBn λ^i-tree searches (the λ^i-trees being of order $i = 0, 1, \ldots, n-1$).

Hence, in order to generate all λ^n-moves at depths 1 and 2, corresponding to the three black moves and the nine white answers in the λ^n-tree in figure 11, $(1+b)Bn$ number of λ^i-trees must be analyzed (the λ^i-trees being of order $i = 0, 1, \ldots, n-1$), all of them to depth $d-2$. By a similar argument, the λ^n-moves at depths 3 and 4 in the λ^n-tree require that $(b^2+b^3)Bn$ number of λ^i-trees (the λ^i-trees being of order $i = 0, 1, \ldots, n-1$) are searched to depth $d-4$, and so on.

Now the question is how deeply the λ^n-tree needs to be searched? This depends on circumstances, and the depth will generally be between 2 and $(d-1) - 2(n-1)$ plies. To simplify, we assume the worst case, namely that the λ^n-tree needs to be searched to its maximum depth, i.e., $(d-1) - 2(n-1)$ plies.

The above reasoning leads to the recursive algorithm shown in figure 12 – stated in pseudo-code – for the total number of positions generated in order to search a λ^n-tree with search depth *d*.

The recursion ends with the number of generated moves in the λ^0-tree, since the algorithm knows that a λ^0-tree demands *B* positions to be evaluated. For given values of *B* and *b*, the above-mentioned recursive formula is thus capable of estimating the total number of positions generated in a λ^n-search to depth *d*, where all the λ^n-trees are searched by means of minimax. This estimate can be compared to $(B^{d+1}-1)/(B-1)$, being the estimated total number of positions generated in a minimax-search. Thus, for given *B*, *b*, *n* and *d*, we define the reduction factor as follows: $r = [(B^{d+1}-1)/(B-1)]$ / lambdaMovesGenerated(*n*, *d*).

```
lambdaMovesGenerated(n,d) {
  if(n==0)return B;
  sum=0;
  for(i=2;i<=(d-1)-2*(n-1);i=i+2) {
    for(j=0;j<=n-1;j=j+1) {
      sum=sum+(b**(i-2)+b**(i-1))*B*lambdaMovesGenerated(j,d-i);
    }
  }
  return sum;
}
```

Fig. 12. Pseudo-code for counting the total number of positions generated by a λ^n-search to depth d.

The reader might now ask why alpha-beta is not used instead of minimax? This is only because assuming minimax makes the reduction factor easier to compute – the point being that using alpha-beta (or any other search technique) instead of minimax would not alter the reduction factor, assuming that alpha-beta (or any other search technique) works equally efficiently in the λ-trees and in the standard (full) search tree. The efficiency of alpha-beta-pruning thus cancels out in the numerator and denominator of the reduction factor, so that the reduction factor can also be interpreted as the efficiency of the λ-search technique (using standard alpha-beta in the λ-trees) over standard alpha-beta.

To provide an idea of the magnitude of the reduction factor, we show two tables below: namely (b, B) = (4, 40) and (b, B) = (8, 40), calculated by means of the algorithm in figure 12.

Table 1. Reduction factor, λ-search relative to standard alpha-beta, (b, B) = (4, 40)

	n=1	n=2	n=3	n=4	n=5
d=3	8				
d=5	772	65			
d=7	76.942	3.174	482		
d=9	7.692.425	209.432	16.192	3.328	
d=11	769.231.503	15.625.505	812.550	88.201	21.791

Table 2. Reduction factor, λ-search relative to standard alpha-beta, (b, B) = (8, 40)

	n=1	n=2	n=3	n=4	n=5
d=3	4,6				
d=5	112	20			
d=7	2.804	251	76		
d=9	70.112	4.173	666	254	
d=11	1.752.804	78.158	8.530	1.810	781

In order to have something to compare with, the reduction factor of standard alpha-beta relative to minimax is shown in the table below.[7]

Table 3. Reduction factor, standard alpha-beta relative to minimax, $B = 40$

	good move order	average move-order	bad move-order
d=3	15	3,9	1,8
d=5	60	7,8	2,4
d=7	238	15,6	3,1
d=9	952	31,2	4,2
d=11	3.810	62,4	5,6

Tables 1 and 2 show that the largest reductions are found for $n = 1$, i.e., if the problem can be solved by means of a number of *direct* attacker's threats (chess mating: checks – Go block tactics: ataris). For growing depth, this renders a huge reduction factor relative to standard alpha-beta, both if the threats and threat-aversions on average make up 10% of the legal moves (table 1), or 20% (table 2). This corresponds to the full search tree being highly non-uniform in a way exploitable by λ-search. But even for more "saturated" problems; i.e., problems needing a larger value of n, the reduction factor is still impressive. For instance, with b/B being 20% as in table 2, a λ^2-search to a depth of 5 plies would still – on average – require only about 1/20 of the positions generated with standard alpha-beta to depth 5, this reduction factor being comparable to the reduction factor of good or average move-ordering alpha-beta over minimax at depth 5 (cf. table 3).[8]

As it is seen, the relative reduction in search space by using λ-search instead of standard alpha-beta is comparable to the relative reduction in search space by using alpha-beta instead of minimax, especially for low saturation problems with $n < (d-1)/2$. It is possible to prove that – for large B and b, and for n relatively small compared to $(d-1)/2$ – the reduction factor, r, can be approximated by:

$$r \approx \frac{\left(\frac{B}{b}\right)^{d-n-1}}{\binom{(d-3)/2}{n-1}}, \qquad \binom{x}{y} = \frac{x!}{y!\,(x-y)!} \quad . \tag{1}$$

In that case, the reduction factor depends on the size of B compared to b in the $(d-n-1)$'th power, divided by a binomial coefficient. A possible interpretation: Assume that we had some divine knowledge, knowing in each position of the full search tree

---

[7] Note: In the calculation of these reduction factors it is assumed that the best attacker's move is found after 10 tries (good), 20 tries (average), or 30 tries (bad). So for instance, for $t = 3$, the reduction for good move-ordering is calculated as follows: $(1+40+40^2+40^3)/-(1+10+10\cdot40+10\cdot40\cdot10)$. It should also be noted that the move-ordering can often be improved upon in a number of ways, but here we focus on *standard* alpha-beta only.

[8] This could, for instance, be a model of those mate-in-two problems in chess where the first attacker's move is not a check (whereas the next is the checkmate). In that case – a λ^2-tree searched to depth 5 – a reduction factor of about 20 would be expected. Alternatively, if the first attacker's move is a check (and the next the checkmate) – a λ^1-tree – the reduction factor would be expected to be about 112.

which b out of the total of B legal moves contain the best move (thus in each position having only to consider b instead of B moves). In that case, the reduction factor would be $r = [(B^{d+1}–1)/(B–1)] / [(b^{d+1}–1)/(b–1)] \approx (B/b)^d$. However, since we do not have access to such divine knowledge, lower-order λ-trees have to be searched in order to find the b interesting moves (this being more costly for higher n), explaining the binomial coefficient and the fact that the exponent is $(d–n–1)$ rather than just d.

From the formula it is seen that the crucial factor for the reduction factor is the *relative* size of b in relation to B. Thus, we would expect similar reduction factors (tables like table 1 and 2) for $(b, B) = (8,40)$ and, e.g., $(b, B) = (50,250)$, whereas the reduction factors for, e.g., $(b, B) = (8,250)$ would be much larger than for $(b, B) = (8,40)$. It should, however, be emphasized that all the above calculations must be taken with a large pinch of salt, since the calculations just yield some theoretical and not empirical indications of the strength of λ-search over standard alpha-beta. A more authoritative estimate of the efficiency of λ-search would imply analyzing a large number of realistic game-problems with different search techniques, including λ-search, thus being able to offer some real-world statistics on the relative merits of the different approaches.

3.2 Null-move Pruning

Alpha-beta augmented with null-move pruning (see e.g. [5] for an overview) might seem similar to λ-search, but even though the underlying idea is quite similar, there are a number of important differences – apart from the fact that alpha-beta augmented with null-move pruning can be directly applied to non-binary search trees (in contrast to λ-search):[9]

First, in λ-search, only the defender makes null-moves, and λ-search does not use depth-reduction when searching the lower-order λ-trees.

Second, a λ^n-search controls the total number of admissible defender's null-moves in any branch between some node and the root, this number always being $\leq n$. Knowing the concrete λ-order of a problem (i.e, distinguishing clearly between different orders of threats) often eases the implementation of abstract game-specific knowledge.

Third, because null-move pruning operates with a depth reduction factor for searching the sub-tree following a null-move, an erroneous result might occasionally be returned. Null-move pruning with depth-reduction works extremely well in many applications (for instance computer chess), but it cannot offer the same absolute reliability as λ-search (cf. the theorems of confidence and convergence).

---

[9] The reader might ask whether λ-search could somehow be used for non-binary search trees? A possibility would be to define the goal as follows: If a leaf-node has evaluation value larger than some threshold, the value of the node is 1, otherwise 0. This way, λ-search could be used in the same way as a null-window alpha-beta-search. However, a general non-binary search tree is usually very uniform (unless, for instance in chess, a hidden mate exists), implying $n = (d–1)/2$ and hence a limited reduction factor (full "saturation"; cf. the diagonals of tables 1 and 2).

Finally, λ-search is not tied to any specific search-technique for searching the λ-trees, thus opening up the possibility of easily combining null-moves with for instance proof number search.

3.3 Proof-number Search

A recent and very popular search technique for searching two-valued search trees is the so-called proof-number search (see e.g. [1] and [2]). Like λ-search, proof-number search exploits non-uniformity of the search tree, trying to search the thinner parts of the search tree first. Still, the differences between the two approaches should be pointed out – apart from the fact that proof-number search has no problems with zugzwang-motives (in contrast to λ-search):

First, proof-number search uses a large working memory overhead, since the whole (or at least much of the) search tree needs to be stored in memory. In contrast, λ-search using, for instance, standard alpha-beta (without transposition tables etc.) as the "search-engine" has negligible working memory requirements.

Second, it is not an easy task to incorporate transposition tables into proof-number search. In contrast, if alpha-beta is used to search the λ-trees, transposition tables can be used with as little difficulty as in standard alpha-beta.

Third, λ-search depends on some technique for searching the λ-trees. But the concrete "search-engine" could be anything as long as it works – including proof-number search. Hence, proof-number search could be combined with λ-search with very little effort (thus combining null-moves and proof-numbers), and even if some new powerful tree-searching technique should be invented in the future, λ-search would most likely benefit from it as well.

Finally, and very importantly, as mentioned in the comparison with null-move pruning, the possibility of distinguishing clearly between different orders of λ^n-trees – i.e, in each position knowing the context of the value of *n* (and in addition: having access to information generated in lower-order λ-trees) – often makes it easier to implement abstract game-specific knowledge. Additionally, the distinction between different orders of λ-trees also makes it possible to concentrate time- and/or memory-expensive techniques on the highest-order λ-trees. Time-expensive techniques could e.g. be elaborate move-ordering schemes or pattern-matching, and memory-expensive techniques could be the use of transposition tables, or the use of proof-number search.

4 Implementing Abstract Game-Knowledge (Go Block Tactics)

4.1 The Relevancy-zone

One of the hardest problems in Go is how to limit the search in open-space Go block tactics by means of characterizing some a priori zone of *relevance*; that is, some area of the board outside of which the addition of any number of black or white stones in any configuration cannot alter the status of the problem. For instance, white *a* in figure 3 cannot be caught in a normal ladder, an example of which is shown in figure 2. Adding a free black stone at *e* in figure 3 will not alter the status of the ladder (still

not working), whereas a free black stone at e.g. *f* will make the ladder work. Actually, adding a free black stone at any of the marked points in figure 6 makes the ladder work; hence each of these points is a possible λ^2-move (loose ladder move) for the attacker.

However, it does not take long looking at figures 2 and 6, before one realizes that the true relevancy-zone seen in figure 6 should be somehow derivable from or dependent upon the "shadow" of the stones played in figure 2. So the idea is that the true relevancy-zone, called the R-zone, is situated inside another zone, called the R^*-zone, consisting of all stones (= "shadow stones") played to prove that a lower-order λ-search (an ordinary ladder) cannot kill the white stone in figure 3. If all the moves of these failing ladders (of which one of them is shown in figure 2) are recorded (circles), and all points adjacent to the circles (liberties of the shadow stones) are added as well (squares), the result is seen in figure 13:[10]

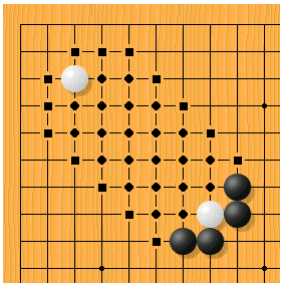

Fig. 13. Relevancy-zone for black?

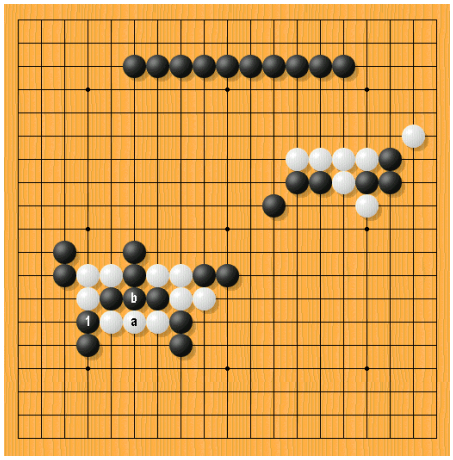

Fig. 14. Is white *a* dead after black 1?

Definition 3. A *shadow stone* corresponding to a λ^n-tree is a stone that is played in this λ^n-tree, or in one of the lower-order λ-trees called from the λ^n-tree.

Comparing figures 6 and 13, it is seen that the R-zone of figure 6 is contained within the R^*-zone of figure 13. This seems simple enough, reducing the number of points considered for finding the attacker's λ^2-moves from 355 to 36. Using this methodology, a point around *e* in figure 3 *will* never be considered as a λ^2-move, whereas a point such as *f* would. There is a problem, however, namely what to do with so-called *inversions*?

[10] Note that in this and the following examples, there is no limit on the maximal search depth. With limited search depth, the relevancy zones would be smaller. Also, note in figure 13 that the liberties of the white ladder-breaker are added (more on this in section 4.2 on inversions).

4.2 Inversions

In Go block tactics, the attacker tries to confine the defender, whereas the defender tries to break loose or make two eyes. The defender can break loose by either moving/extending out, or trying to capture some of the attacker's surrounding stones. We denote the defender's trying to catch some attacker's stones an *inversion*, since the roles are switched.

To yield an example, we consider the defender disturbing/averting the threat of a loose ladder (λ^2-tree) by means of an (inverted) ordinary ladder (λ^1-tree). An example is shown in figure 14.

Black has just played 1, a λ^3-move threatening to kill white *a* in a λ^2-tree (loose ladder). Black 1 is a standard tesuji for capturing a block like *a*, but before dooming *a* dead, we need to consider the fact that the surrounding black block *b* is vulnerable due to a limited number of liberties. Hence, the question is: what are the possible white λ^3-moves following black 1?

In order to answer this, we play out the threatened λ^2-tree, killing white. This is shown in figure 15.[11] These moves can also be found as circles in figure 16.[12] Since the black block *b1* has three liberties only, these liberties are also added to the R^*-zone. The general rule for doing this makes use of the concept of quasi-liberties, defined below.

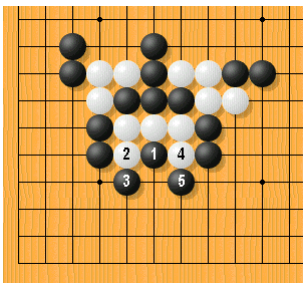

Fig. 15. Killing white in a λ^2

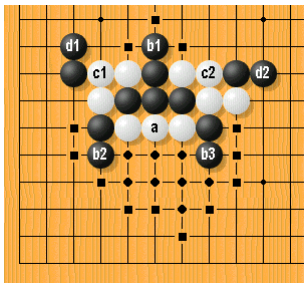

Fig. 16. Relevancy-zone for white

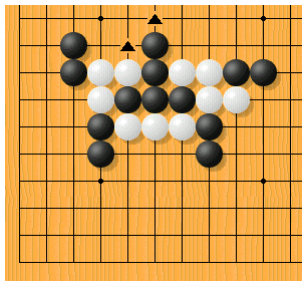

Fig. 17. White λ^3-moves

Definition 4. A *quasi-liberty* corresponding to a λ^n-tree is a liberty that is not coincident with a shadow stone corresponding to the λ^n-tree. (The number of quasi-liberties will always be ≤ the number of real liberties)

Example: In figure 16, the block *b1* has 3 quasi-liberties (coinciding with its real liberties since none of the points surrounding *b1* are circles (shadow stones)). The blocks *b2* and *b3* have 3 and 2 quasi-liberties, respectively.

---

[11] The sequence in figure 15 contains both λ^2-, λ^1- and λ^0-moves (black 1 is a λ^2-move, black 3 is a λ^1-move, and black 5 is a λ^0-move), since all moves in any lower-order λ-tree are "recorded" to yield the shadow stones.

[12] The reason why there are 8 circled moves in figure 16 compared to the 5 shown moves in figure 15 is that the shadow stones in figure 16 (circled moves) also contain white moves played at the three extra points. These moves are (non-working) white escape-attempts.

Using the concept of quasi-liberties, the inversion rule can be stated:

Inversion rule: Consider a λ_a^n-tree with value 1 or 0. Call the defender's block a 0-surrounding block. Now, repeat the following for $m = 1, 2, 3, \ldots, \infty$: Find all blocks of opposite colour touching the already found (m–1)-surrounding block(s). Of these new potential m-surrounding blocks, only keep those that have not been found already for smaller m, and for which the following is observed: $q < n - m + 3$, where q is the number of quasi-liberties of the block.

Given this, the R^*-zone corresponding to the λ_a^n-tree can now be defined. First, take the shadow stones and points adjacent to the shadow stones. Next, if $\lambda_a^n = 1$, all quasi-liberties of all surrounding blocks from the above list of the *same* colour as the attacker are added to the R^*-zone. Conversely, if $\lambda_a^n = 0$, all quasi-liberties of all surrounding blocks of the *opposite* colour of the attacker are added to the R^*-zone.

Example: All attacker's blocks touching the defender's block and obeying the equality $q < n - m + 3$ are called 1-surrounding. In the example, we are analyzing the relevancy zone of a λ_a^2-tree with value 1 (in order to find λ^3-moves), and hence $n = 2$. Thus, in order for black blocks to be 1-surrounding, they must have fewer than $n - m + 3 = 2 - 1 + 3 = 4$ quasi-liberties. In figure 16, it is seen that *b1* has 3 quasi-liberties, whereas *b2* and *b3* have 3 and 2 quasi-liberties, respectively. Hence, both *b1*, *b2* and *b3* are 1-surrouding blocks.

The blocks *b1*, *b2* and *b3* touch three white blocks, *a* already being in the list, and *c1* and *c2* being new. In order for a white block to be 2-surrounding, it must have less than $n - m + 3 = 2 - 2 + 3 = 3$ quasi-liberties. Since *c1* and *c2* have 3 and 4 quasi-liberties, respectively, these blocks are not 2-surrounding. As there are no 2-surrounding blocks, the search for m-surrounding blocks is terminated, not worrying about the potential 3-surrounding blocks *d1* and *d2* (even if they had had less than 2 quasi-liberties).

The true R-zone corresponding to the λ_a^2-tree is shown in figure 17, where it is seen that only two white moves actually disturb/avert the threatened λ_a^2-tree of figure 15 (all other white moves fail to disturb figure 15). The two triangled white moves disturb the λ_a^2-tree because they render possible the inverted ladder shown in figure 19 – an inverted ladder black has no way of escaping.

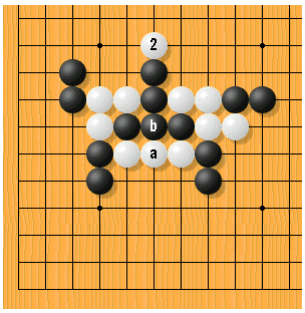

Fig. 18. Does white 2 save white *a*?

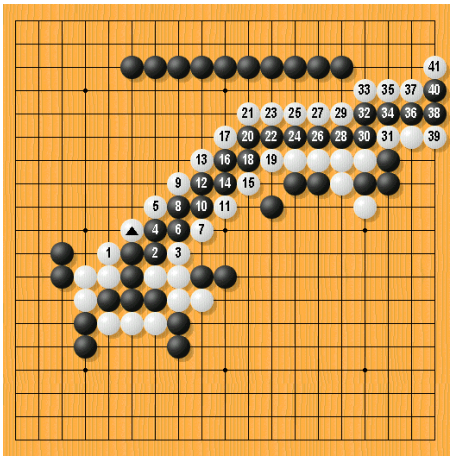

Fig. 19. The threatening inverted ladder

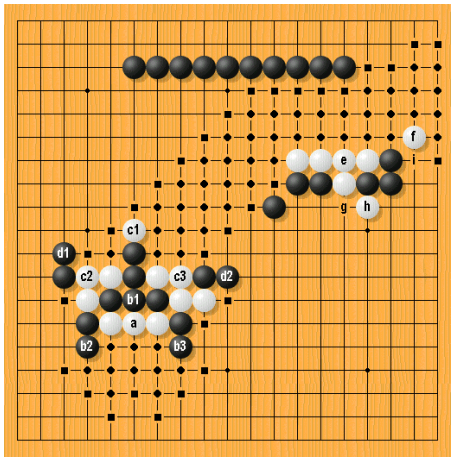

Fig. 20. Zone made by the inversion rule

Now consider figure 18, where white has just played the λ^3-move white 2, disturbing the threatened λ_a^2-tree of figure 15. To find candidates for the next black λ^3-move, the now failing λ_a^2-tree is played out, and all those λ^2-moves (and the λ^1- and λ^0-moves generating those λ^2-moves) are recorded. The shadow of these stones is shown as circles in figure 20.

Using the inversion rule, the relevancy zone would look like figure 20. This seems to be a safe bet for a R^*-zone containing the true R-zone of black disturbance-moves (black λ^3-moves following white 2 in figure 18). However, the inversion rule is not 100% safe, which is seen by the fact that a black stone at *g* is actually a disturbance threatening $\lambda_a^2 = 1$. This is so, because after black *g*, the inverted ladder shown in figure 19 no longer works – and the example is constructed in such a way that after black *g* there is no other ladder that works for white. Thus, black *g* should in principle be counted among the possible black λ^3-moves in this position, and thus be part of the R^*-zone.

Obviously, *g* is not the best move for black to play in order to fend off the threatened ladder of figure 19, but we are interested in ensuring that *all* possible black λ^3-moves are generated, leaving it up to some successive forward-pruning heuristic to cut some of them off afterwards, if they can safely be judged inferior to other black moves. But at least we should know that moves like black *g* exist.

A much better black λ^3-move would e.g. be a move at 1 in figure 19. After such a move, white is close to being dead, since white will soon run out of (inverted) ladder-threats on the black block *b1*. But the interesting thing about a black move at *g* is that it encloses the territory below *g* at the same time as threatening to kill *a*. Hence, black *g* could be used as a forcing territory-enclosing move or as a ko threat.[13]

The question is how the R^*-zone catches a move like black *g*? The answer is that the white block *e* is actually surrounding the black block *b1* in much the same way as

[13] Interestingly, a white λ^3-answer to a possible black λ^3-move at *g* in figure 20 could be a move directly to the left of or below *g* (giving an atari on the black stone at *g*). This would be yet another inversion.

c1, *c2* and *c3*, even though it is, of course, much more indirectly. It *is* possible to formalize this with help of additional concepts of so-called quasi-surroundedness (white *e* quasi-surrounding *b1*) and so-called shadow blocks, but space does not permit going further into this here.[14]

Apart from the remaining problems of the inversion rule, however, this rule at least seems to contain some of the right abstract concepts for the construction of reliable relevancy-zones, even though it may still overlook a few exotic moves. Such exotic moves are typically easy for the opponent to fend off and usually not relevant to the solution of the problem.[15]

More work needs to be done in this field, and until some algorithm for the construction of R^*-zones can be proved to contain the true R-zones in all cases of Go block tactics, an algorithm like the proposed inversion rule would just have to be seen as providing some useful foundations for the implementation of abstract topological knowledge regarding open-space Go block tactics. Perhaps some kind of automated theorem-proving could be of use here; cf. the interesting approach in [4]. And for another example illustrating the inversion rule, the reader is referred to [11].

5 Conclusions and Scope for Further Work

To conclude briefly, the λ-search method seems to be very well suited for obtaining well-defined goals in two-player board games like chess or Go, provided that passing is allowed or zugzwang is not a motive. As shown in section 2, λ-search offers the theorems of confidence and convergence, and the algorithm is simple and requires negligible working memory in itself. As section 3 indicates, in many cases λ-search is capable of offering a relative search space reduction over standard alpha-beta comparable to the relative reduction of standard alpha-beta over minimax. In addition, the λ-search method often eases the implementation of abstract game-specific knowledge, and λ-search can be combined with any search method for searching the λ-trees, including proof-number search (thus rendering possible an easy combination of null-moves and proof-numbers).

In Go, the λ-search algorithm is capable of solving open-space tactical problems (tesuji) from e.g. volume 4 of *Graded Go Problems* [6] with relative ease.[16] Regarding the scope for further work, the following points could be considered:

---

[14] With these additional concepts it is also possible to justify the inclusion of the liberties of the white ladder-breaker in figure 13 in the R^*-zone.

[15] As noted before, such moves could still be relevant as forcing moves or ko threats. Also, in the context of finding double threats, such moves can be highly interesting.

[16] As an example, the following numbers have been tried out successfully: 2, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 41, 103, 104, 105, 107, 108, 110, 113. The program uses iterative deepening alpha-beta with transposition tables for searching the λ-trees, together with the inversion rule described in section 4. In addition, moves in all λ-trees are ordered by counting the number of liberties and meta-liberties (points adjacent to liberties) of the attacked block after each possible move (minimizing these if the attacker moves, and maximizing these if the defender moves, and also prioritizing moves with low Manhattan distance to the attacked block).

Generally:

- Can the algorithm be taught to find at least one of two (or more) goals by means of identifying double-threat-moves?
- Which kinds of search techniques are well-suited for λ-trees? How to manage the use of transposition tables in the different orders of λ-trees? Which kinds of move-ordering techniques could be useful (iterative deepening with transposition tables, history heuristic etc.)? Use of proof-number search to solve the λ-trees?
- Investigate a large number of realistic game-problems with λ-search and other search methods, in order to provide some real-world statistics on the efficiency of the different approaches.
- Can zugzwang-motives somehow be incorporated into λ-search?
- How to implement parallelism (multiple processors) most conveniently into λ-search?

Go-specific

- Go block tactics: Solving *inversions* (cf. section 4.2) as independent sub-problems (local games) if possible. Perhaps some of the tools described in [8] could be of use. Try to reduce the size of the R^*-zone as much as possible by means of solving all λ^n-trees two times – the second time using the best moves (stored in a transposition table) from the first search. Try to classify certain kinds of λ^2-moves, see e.g. [7].
- Implementing abstract knowledge regarding tsume-go, semeai and connection problems in the context of the λ-search methodology.
- Using λ-search to search for eyes or Benson-immortality, see [3] and [9].
- How to handle seki or ko?

References

1. Allis, L.V: Searching for Solutions in Games and Artificial Intelligence. PhD Thesis, University of Limburg, Maastricht (1994)
2. Allis, L.V., M. van der Meulen, and H.J. van den Herik: Proof-Number Search. Artificial Intelligence, Vol. 66, ISSN 0004-3702 (1994) 91-124
3. Benson, D.B.: Life in the Game of Go. Information Sciences, vol. 10 (1976) 17-29
4. Cazenave, T.: Abstract Proof Search. In I. Frank and T.A. Marsland (eds.): Computers and Games 2000, Lecture Notes in Computer Science, Springer-Verlag (2001)
5. Heinz, E.A.: Adaptive null-move pruning. ICCA Journal, Vol. 22, No. 3 (1999) 123-32
6. Kano, Y: Graded Go Problems for Beginners, Volume Four, The Nihon Ki-in, Tokyo, Japan (1990)
7. Kierulf, A.: Smart Go Board: Algorithms for the Tactical Calculator. Diploma thesis (unpublished), ETH Zürich (1985)
8. Müller, M.: Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory. PhD thesis, ETH Zürich, 1995. Diss. ETH Nr. 11.006 (1996)
9. Müller, M.: Playing it safe: Recognizing secure territories in computer Go by using static rules and search. In H. Matsubara (ed.): Game Programming Workshop in Japan '97, Computer Shogi Association, Tokyo, Japan (1997) 80-86

10. Müller, M.: Race to capture: Analyzing semeai in Go. In Game Programming Workshop in Japan '99, volume 99(14) of IPSJ Symposium Series (1999) 61-68.
11. Thomsen, T.: Material at http://www.t-t.dk/go/cg2000/index.html (2000)
12. Wolf, T.: About problems in generalizing a tsumego program to open positions, 3rd Game Programming Workshop in Japan, Hakone (1996)

Appendix: λ-search illustrated on Chess

Consider the mating problem in figure A1. In chess mating problems, a λ^1-move is a check, but it is easily seen that the black king cannot be mated with a sequence of white check-moves. Hence, in this problem, $\lambda_a^1 = 0$. However, the black king can be mated in a λ^2-tree, as shown below.

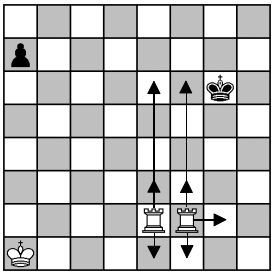

Fig. A1. How to mate black? White λ^2-moves

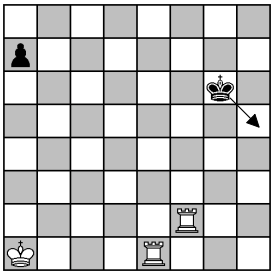

Fig. A2. Black λ^2-move following Re2-e1

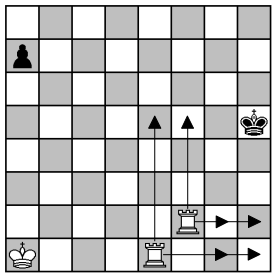

Fig. A3. White λ^2-moves following Kg6-h5

In figure A1, all the white λ^2-moves (moves threatening a forced mating check-sequence) are shown. Of these 7 white moves, three of them are direct checks, in contrast to the four other more "quiet" moves (Re2-e3, Re2-e1, Rf2-f3 and Rf2-f1). For instance, after white Re2-e1, if black passes, a mate results from Re1-g1+, Kg6-h7/h6/h5, Rf2-h2++. Now, consider Re2-e1. As shown in figure A2, there is only one black λ^2-move averting the threatened mating check-sequence: Kg6-h5. After Kg6-h5, white has no mating check-sequence. His best try would be Re1-h1+, Kh5-g4, Rh1-g1+, but it peters out after Kg4-h3. In figure A3, the 6 white λ^2-moves following Kg6-h5 are shown. Of these, four are direct checks, while the more quiet Re1-g1 and Rf2-g2 threaten mating check-sequences. After either Re1-g1 (or Rf2-g2), black cannot avoid a mating check-sequence no matter where he moves, implying that there are no λ^2-moves following Re1-g1 (or Rf2-g2). Hence black can be declared dead.