The program `GoTools` and its computer-generated tsume go database *

Thomas Wolf
School of Mathematical Sciences
Queen Mary & Westfield College
Mile End Road
London E1 4NS
email: T.Wolf@maths.qmw.ac.uk

October, 1994

Abstract

In this talk the program `GoTools`, designed for solving life/death problems in Go, will be described and examples will be given. Remarks are made

- to the kernel modules for solving problems,
- to new features which increase the programs utility value like a playing- and an exercise modus,
- to the problem generation capability and the database of over 40.000 generated tsume go problems,
- to current limitations of the program.

Example positions are given in several sections including faulty problems from the literature which `GoTools` solved correctly.

1 About the program `GoTools`

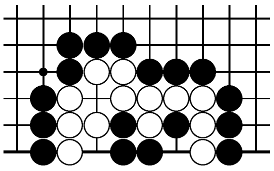
1.1 Contents of the problem solving part

The core of the program, the problem solving modules have been partially described earlier in [8]. The following are the important ones.

- A module which provides an ordering of the usefulness of moves on a heuristic basis. If there is one or more winning moves in a given situation then such a move comes currently on average on better than second place in the heuristic ordering.

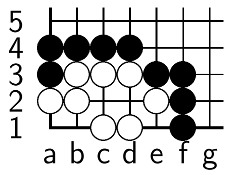
*Published in the Proceedings of the First Game Programming Workshop in Japan, Hakone Oct. 1994, p. 84-96.

- A module for directly detecting (i.e. without doing case distinctions) whether the life of a position can be ensured by only the following means:
  - capturing attacking chains with only one liberty. Example:

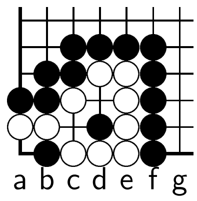


secure life

  - uniting two securely linked chains if the empty linking field(s) is(are) attacked. If, for example, in the following left diagram ○on e2 would be attacked then it can link safely on e1 or d2 without filling an eye. More difficult is to see when this is not enough as in the following right diagram. Also here ○on c3 can link safely with c2 or d3 without filling an eye but still ○is not alive unconditionally.



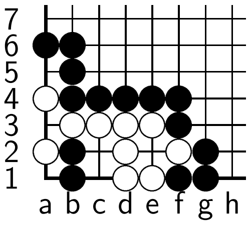
secure life



no secure life

- A module for classifying the state of an eye which is surrounded by only one chain and which is not big enough to include an enemy eye. This module works algorithmicly and is very likely slower than a similar module described by Dave Dyer [1] which stores all 560523 possible shapes of eyes of size 7 or less in a database. On the other hand the algorithm needs very much less space to allow the program to run on simple PC's with 640kB.

- A module for the direct detection of secure death without doing case distinctions. Example:



- A module for catching a chain with two liberties.

- A module for recognizing secure links of a number of chains through either at least two common liberties or one common secure liberty.

- A module for organizing the tree search including some forms of dynamic learning during the search.

- A module to provide a hash database which is able to work with ko-specific results.

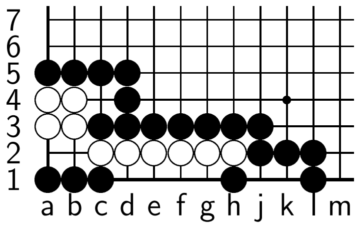
1.2 Remarks to the tree search

An excellent explanation of life/death searching in go has been given by Dave Dyer on the computer go email list [2]. As here is not the place to repeat most of it, only a few remarks shall be made.

As mentioned below, an α/β-tree search with only two states 'alive' or 'dead' is performed where seki is treated the same as life. Single and multiple ko's are recognized, see below. Sequences of moves in this search are stopped by modules for recognizing life and death as described above. Ways to make the search more efficient are the following.

- If during a search a move A is counter proofed by a move B then the next move that is tried instead of A is the move B if this is legal and has not already been tried.
- Regarding the tree of moves as a family tree then moves which already have been successful will get a high ranking in other closely related situations, e.g. as cousin.
- A hash database stores intermediate results. It stores whether a specified color that is moving next, will win or lose, given a number of external threats for one or the other side. The result is supposed to be exact.
- Some moves like self-atari moves with more severe restrictions are forbidden, but see below the section about 'Safety first'.
- A number of parameters provide arbitrary tree cuts. These usually speeds up the search considerably but also have the danger of overlooking the correct line of play. That this is a human problem as well is shown in the appendix.

The methods described above allow the solution of non-trivial problems in a reasonable time as the following example shows. This problem was shown to the author by Denis Feldmann.



● to move first and get a ko in which ● captures first.
`GoTools`: ● on `f1` (12.6 sec)

All times given are measured on a PC486 (33MHz) if not otherwise stated. As far as the author knows, is `GoTools` currently (Sep. 1994) the fastest tsume go program for medium and hard problems (for other programs specialized in tsume go see [3] – [7]).

Whereas the kernel modules for solving a problem did not change much over the last year, some new features were added to make the program more user friendly.

1.3 Extra features

Apart from the pure solving mode of problems there are three more modes of operation available.

One of them is the Play-mode in which one can play a problem directly against the computer. If the problem has been solved by the program before then it will answer instantly as all results of the search tree have been stored in a hash database. Otherwise the program needs a few seconds at the start (or more if the problem is harder) but will answer immediately afterwards. *The ability to play against the computer directly (even difficult problems with a quick or reasonable response time) adds a new dimension to learning to fight life/death problems. It is a qualitative improvement on printed material where the number of displayed variations is very limited.* This is especially unsatisfying if the book can not counter-proof any 'new solution' the reader has found. Also different to a book, the correctness of the program can be checked by the program itself through playing against it and switching the 'Comment flag' on. If the program is counter-proofed due to an internal error then the program realizes that the outcome differs from earlier predictions. It then admits the error and asks to send this problem to the author.

The two other modes concern the automatic generation of new problems in the mode 'daily fresh' and a training game with problems from a database called 'daily exercise'. Both are described in later sections.

1.4 Watching the progress

After the source code has grown to a few hundred kByte with many parts having been partially rewritten and worked on repeatedly it is getting more and more difficult to improve the code and even to know what the program actually does. Though (in a private version) there is the opportunity to go slowly step by step through the calculation, a long search takes minutes during which millions of lines of code have been executed which to go through in debugging mode is impossible. What is needed is a more global look at what is going on during the search with the possibility to look closer if necessary.

A way to realize this is to display in graphic mode the search depth (vertically) as a function of time, or more precise as a function of the number of leaves of the search tree investigated so far (horizontally). Coloring moves of the attacker and defender differently gives a quick visual impression of who is winning how often currently and where in the tree search something looks suspicious, like a sudden increase of the search depth by many moves. This can happen if both sides played silly moves. That one side (the loser) plays silly moves is normal as the loser has to try all possible moves, also silly ones, to be sure about the loss. But if both sides play bad moves then this is a case for improving the heuristics.

One improvement that resulted from the graphical overview was that in problems where the defender wins there are many leaves in the tree where the position lives and where the killer had to try many useless moves to prevent the live. It turned out that one can somewhat cut down the number of moves of the killer to prevent live by analyzing how the defender lives and to drop obviously useless killer moves.

A further way to watch what is going on is the display of the first five moves of the sequence of moves currently investigated. This display is more to entertain than for improving the code.

1.5 Time estimates

An annoying aspect of 'watching the progress' is the uncertainty about how long the calculation still might take, a minute, an hour or a year? Recently a possibility for estimating the time still necessary to solve the problem has been added. It turns out that a good estimate is to take the minimum of two numbers, each being the maximum time for either side to win. To estimate this maximum time it is assumed that the number of possible moves depends on the depth of search but is constant for a given depth (which is of course not true). The remaining time is then determined on the basis of the time already spent and the size of the portion of the tree already searched compared with the size of the tree still to be searched in the worst case. On one hand this estimate is too big because the branches of the tree tend to get smaller in time, as more and more silly moves that are tried by the loser can be counter-proofed more and more quickly by the winner. On the other hand the hash database becomes less effective because of its limited size, keeping a smaller and smaller portion of the solved tree for later use.

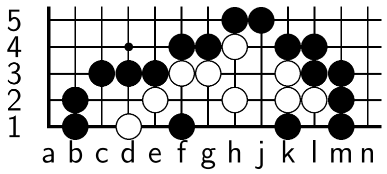
The time estimated turns out to be correct within a typical factor of 1.5 - 2. This is rather good given that the time may range over many magnitudes (sec - days). To get a feeling about how much time estimates can fluctuate at the start of the solution and how they stabilize later, the estimated total time and the time already spent are displayed graphically as well in the above mentioned diagram.

2 A database of tsume go problems

2.1 The generation of the database

Once an effective l/d-program is available, it is not difficult to add a tool which generates problems automatically. This can be done by strengthening the loser in a randomly generated position until each side reaches some success if it moves first. The original position is obtained by extending randomly the area around one of the points A1 or K1 or K10 to obtain a problem at the corner, the edge or in the middle of the board. This area is enclosed by a secure wall of one color and the other side starts setting stones till it wins or reaches ko. To increase variety the 'killing' side is also allowed to make moves occasionally. To speed the process up, the time consuming exact search is only done if equal strength is reached.

From the resulting situations only those are kept where at least one side has exactly one best move to win or at least to reach an optimal ko. At first each problem is stored in a single file. To save disk space, each 256 generated problems (generated in slightly more than a day on a PC 486 with 33 MHz) are packed and stored in a single file. That there are not only easy ones among the generated problems shows the following example.


● to move first and get a ko
in which ● captures first.
`GoTools`: ● on `g1` (307 sec)

For applications to be described below, the raw randomly generated problems have to go through a refinement procedure.

2.2 Refinement of the database

Because the database has been generated over about 3 years with many program errors found in the meantime, it was advisable to solve the problems again with the latest version of the program. For refining them, at first unnecessary stones outside of the 'inner area' of each problem are dropped, i.e. the problems are cleaned up. Then the major problem is to identify and drop similar problems. This is done by rotating, mirroring and shifting them into a unique position. As criteria for selecting the unique rotation, reflection and shift serve the fields around the winning move as those are the most crucial fields of the problem. Criteria for identifying problems are the following.

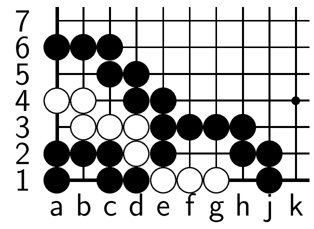
- The same status including the same type of ko if the status is ko.
- Identical first moves in the winning sequence of moves. Only the loser may vary in his(her) response in single moves or the order of two pairs of successive moves may be reversed.
- The following characteristics must be comparable:
  - The size of the verification tree V, i.e. the size of that tree which results if in each depth the winning move would be done first, i.e. if the heuristic would be perfect.
  - The ratio of the size of the actually searched tree and V, i.e. the unnecessary overhead due to an incomplete heuristic or strange properties of the problem.
  - The place on which the winning move landed in a first heuristic which `GoTools` does before solving the problem.
  - The number of moves of the winning sequence of moves.
- Only 4 fields of the inner problem area may differ between two similar problems.

To have not to compare each of the 40,000 problems with each other one, the problems are sorted at first into disjunct sets. These are corner-, edge or middle-board problems and a further criterion is, which of both sides, or possibly both sides, have a unique winning move. This gives 3x3 classes. All problems of each class are then ordered by the size of their verification tree to have a good chance to find identical problems in the neighbourhood in this ordered list. It turns out that the criteria given above are enough to make an automatic selection possible. Only in rare cases the above criteria contradict each other and then the two compared problems are shown on screen and human help is necessary. In this way about 25,000 problems survive. For one of the applications described below the problems have to be ordered according to their difficulty for human players. This 'human difficulty' is modeled by a combination of the above criteria including the number of possible first moves.

2.3 Application of the problem collection

2.3.1 Safety first

Compared with writing a full game-playing program, a life/death program is very much simpler, at least if both play with the same standard. On the other hand even a life/death program can be arbitrarily complex. The bigger the problem, the more moves must be disregarded from investigation and to do this safely is rather difficult. For example, in the following problem the winning move of ● is to fill the own eye which is easily overlooked by humans and by good efficient programs, not by slow programs.



● to move first and kill.
`GoTools`: ● on `b1` (10.4 sec)

The essential problem is therefore to be more and more selective through algorithms and tactical submodules (atari, secure links, status of eyes surrounded by only one chain, ..) and still not to overlook very special cases and situations. It turns out that even after years of debugging `GoTools`, problems turn up which expose a weakness of the program that so far has not been spotted.

Incorrectly solved problems in the literature (see appendix) show that this is not only a problem of programs but also of players of all levels.

Here is, where the database of now over 40,000 problems becomes indispensable. At first one will use tsume go problem books for checking the program. Then one can use masses of computer generated problems to make consistency checks for which these problems even need not be solved correctly. They are:

- The status of the position and the first move (if it is supposed to be unique) must be reproduced after reflection or any rotation.
- The status must be unaltered after doing the winning move and then re-evaluating the position.

Finely one can check the status and first move directly with other tsume go programs. Recently the tsume go database was provided by the author through email to anyone who was interested in using it for computer go. Since then Torsten Harling and Dave Dyer started checking these problems with their own tsume go code. This was and still is quite helpful for improving all involved programs.

2.3.2 Learning from the database

A less direct way of learning from the database was described in [9] where 64 problems have been solved by humans and the computer and the necessary times for different problems have been plotted against each other to spot in which problems the human player was especially fast and in which the computer was much faster than the player. This can help to identify weaknesses of the program heuristics as well as of the player.

For computers to learn automatically other methods like genetic learning and neural networks are popular. Workers in these fields found it especially helpful when the database was made publicly available recently for computer go purposes. Advantages in comparison with learning from games are:

- The optimal move is one out of 5-25 possible moves and not one out of 100-300.
- The motivation for the best move in tsume go is more obvious than in a game because a wrong move is counter-proofed within 5-30 moves and not only after 100 moves or more.
- Only in local problem situations one is really sure what the correct move is. What the best move is in a game may depend on other criteria like the ability to stand through a more risky approach the need to play inaccurately to catch up.

2.3.3 The database for practicing

Depending on how much time one has in a game one may need to assess a position very quickly or one can read it out in detail. Both abilities can be practiced with problems provided from a database. One only has to chose different time limits in the following exercise.

For a given number of problems the first move has to be found. For each problem one chooses the difficulty, then a problem is selected randomly from a set of some hundred problems with that difficulty. Depending on the difficulty one chooses, a different maximal bonus is possible. Full marks are given, if the winning move is found within a first time limit. If not, then the obtainable bonus shrinks gradually to zero within a second time limit. Half the bonus is given, if a move at least provides a ko, in case a better ko or even a straight win would have been possible. Such a module is included since version 1.1 of `GoTools`.

Though new features have been added, there remain some limitations of the kernel program for solving life/death problems which should be mentioned as well.

3 Limitations of version 1.3 of `GoTools`

3.1 Classes of problems

The main restriction is that so far only positions with a completely closed boundary can be investigated. As a consequence, open problems will be closed before evaluation. If this automatic closure provides a boundary which is too wide then the computing time may become unnecessarily high. If the boundary is too tight then the status might be affected by the new extra boundary stones. Then one should close the boundary manually.

Though a semeai situation is different from a usual life/death position, an extended life/death problem may have a semeai (run for liberties) problem inside. Special knowledge for speeding up the solution of semeai is not included so far.

3.2 The solution

The program can determine one winning move as well as all winning moves. To increase efficiency, for the follow up moves always only one winning move is determined, i.e. the moves of the winner in the winning sequence of moves need not be the quickest way to win and the way the looser played need not be the optimal way to postpone the loss.

To improve the play of the losing side in the single sequence that is chosen, an option is available. If this option is on then from all possible moves of the loser those are determined which impose the biggest threat and from these moves one is selected. To determine the threat behind moves of the loser costs little time for simple problems where the search tree fits into the hash data base whereas for difficult problems the computation time may easily be doubled by this extra investigation.

3.3 The treatment of seki

During a tree search also passing moves are tried if no other moves work in a given situation. Though this enlarges the search-tree considerable, it is necessary e.g. in special seki situations to realize that passing is the only correct move.

In the current version 1.1 of `GoTools` there are only two status possible - the position being alive or dead - which means the setting of stones continues until the program sees the position to live safely or to be absolutely dead. The fact to have only two status does increase efficiency, but it also has the consequence that seki has either to be treated as alive or dead. In `GoTools` it is regarded as being equivalent with the position being alive.

For the same efficiency reason also other positions are treated as straight life or death though they are, strictly speaking, not quite the same. Examples are double ko and triple ko. They will be discussed in the following section about ko's.

3.4 The treatment of ko's

3.4.1 Preliminary remarks

If a position is a ko then the outcome of a ko-fight is heavily depending on the rest of the board. On the other hand, the purpose of evaluating a local position is to split the evaluation of the whole board into a number of easier evaluations of parts of the board if that is possible. In tsume go such a causal independence is provided, e.g. if a strong chain of one color surrounds a given area. Another such situation is the endgame when only local fights can bring a few points. In the book of Berlekamp [10] a method is described which is applicable if the local fights are single endgame positions each with just one point to win. If the local fights are about life and death then much more points are on stake and the status of each local fight is more difficult. The question is what are the necessary data which are necessary to describe each life/death fight in order to make a theory for a "sum of life death fights" possible, similar to Berlekamp's theory for a "sum of end-games". The aim is to find a compromise between describing a situation through

- its whole search tree (or the relevant part of it, i.e. the verification tree) or
- describing it just by one of the few labels 'alive', 'dead', 'seki' and 'ko'.

If the status of a position is determined as usually through an $\alpha-\beta$-tree search then during the search each outcome (e.g. seki, ko) must have a given rating such that all outcomes can be ordered and it can be decided which out of two outcomes is better and therefore which of the two corresponding moves is better. But if the status is a difficult ko then such a linear ordering of outcomes is not possible, because whether, for example, a seki is better or worse than a special ko depends on the number of available ko-threats on the rest of the board. Therefore the optimal move and achievable outcome has to be determined for at least two cases, 1) there are enough outside ko-threats or 2) there are not enough threats for that side which would lose if no ko-threats were played. Therefore different $\alpha-\beta$-searches are necessary, in each a different global situation is assumed, i.e. different numbers of ko-threats are available. Another criterion in characterizing ko's is whether one side can afford to pass once or several times in the ko-fight. What of that is realized in `GoTools`?

3.4.2 Different ko's in `GoTools`

The whole procedure is as follows.

At first the program determines the winner if no ko-threat outside of the position is played. If during this run the loser, say player A, would have liked to re-catch a ko (which (s)he did not as no exterior threats have been played) then the whole tree search is done again, now with the loser A having the right to re-catch once, i.e. to violate the ko-rule once in each investigated sequence of moves. If in this second run Player A won, then investigation stops and the situation is, as we shall call it, a '1-threat ko'. If player A lost again and had no chances in the second run to play and win a ko then there is no sense in giving A even more ko-threats. Then the situation is a straight loss for A and investigation stops. If A lost but could have re-catched a ko if (s)he had more exterior ko-threats then a third tree search starts with A being allowed to re-catch twice in each sequence. This is repeated if necessary, to a maximum of 5 re-catches of A. If they are not sufficient then A has a straight loss. This leads to altogether 12 different cases: If a player, say B, moves first then the cases ordered from best to worst for B are:

- a straight win for B,
- a win for B if the other side has less than 5..1 ko-threats otherwise a loss for B,
- a loss if B has less than 1..5 ko-threats otherwise a win for B,
- a straight loss for B.

Here the phrase "B has n ko-threats" means that B wins the ko-fight n times, each time either by the other side

- not having a ko-threat left to force a move of B or
- having a ko-threat but B ignoring it and paying therefore a price.

This distinction of different ko's helps to select optimal moves. Often different moves lead to ko but these ko's are usually not equivalent. The next accuracy level would be not only to count the necessary violations of the ko-rule which are necessary but also how often one side can afford to pass without giving up, like in a multiple step ko. This number of affordable passes is currently not counted.

Finally a short remark about the treatment of double and triple ko's in `GoTools`. As mentioned above, in the tree search of `GoTools` a position can either be alive or dead but not have an intermediate status, i.e. seki is treated the same as life. Similarly with positions like double ko or triple ko which are not quite the same as life or death but which are taken to be equivalent to one of these. How is the status of those life/death problems defined which *necessarily* lead to a repetition of one and the same position when playing it?

The rule used in `GoTools` is as follows. If a player makes a move that leads to an identical situation encountered before (the same position and ko-situation) then the sequence of moves stops and the following rules apply in this order:

1. The side which needed exterior ko-threats in this cycle lost. (Only one side can have violated the ko-rule, the other side already won the fight when no ko-threats were played and therefore will not exercise any exterior threats.)

2. The side which lost less stones in this cycle, or, equivalently had more passes in this cycle, wins.

3. The side which wants to live, has won.

This win or loss only describes the cyclic sequence of moves, not the whole problem. The first rule, for example, says that neither side can win both ko's in a double ko situation. The second rule is obvious and the third rule just implies that everything that can not be captured, like in a triple ko, lives. In this way one may get the result that a position is dead but one does not know whether, for example, a double ko plays a decisive role. This would be of interest if there is more on the board than only the life-death problem. To find this out, `GoTools` has a mode where the status is determined once with the above rules and once with each of the first two rules separately reversed. If the results of both runs differ then a double ko or a cyclic situation like a triple ko is essential for this problem, otherwise not.

3.5 Availability

Information about how to get GoTools is available on the www net under http://www.eng.ox.ac.uk/~syshf/go/t.wolf.html.

A free demo version of the program is available by ftp from lie.maths.qmw.ac.uk (internet number 138.37.80.52). Login as ftp and give your email address as password. The files are in the directory pub/weichi. One either gets gdemo.tar.zip in binary mode or gdemo.tar.zip.UU in normal transfer mode and uudecodes this file afterwards or one gets the 4 files xaa, xab, xac, xad and appends them in this sequence to obtain gdemo.tar.zip.UU.

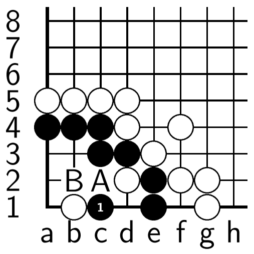
The demo version shows all features of the full program. Its restriction is that it can solve only very simple problems, i.e. also in play mode it makes sense only to play very simple problems. In the 'Daily exercise' mode where in the full version 12,000 or 24,000 problems are available to learn from, only a few hundred of very simple problems are accessible.

4 Appendix

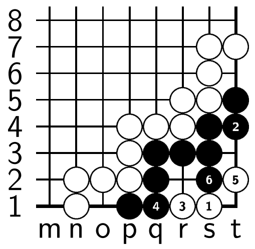
In this appendix examples of wrongly solved problems are given which have been mentioned earlier in the text.

`GoTools` has been used to check the solutions given in a number of well known standard books on tsume go, and in periodicals/calendars. The program found that in all of them which were investigated by the author, there are at least one or more serious mistakes (in about $\frac{1}{2}\% - 2\%$ of the problems) and in up to 10% of the problems there exist other solutions, i.e. other winning first moves, which are not reported.

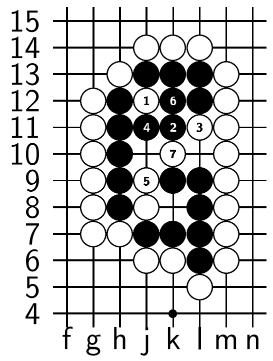
In the following examples the old and incorrect solutions are shown in the diagram.

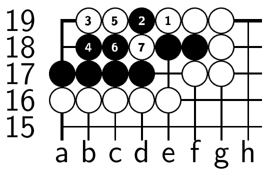


Problem A. [11]
● to move first and live.


Problem B. [12]
○ to move first.
● is alive.

In both first cases the first move is right but not the status, as ko's can be reached by continuing to play from outside. In problem A, ② can play on f1 continued by b2 d1 c2 a2 a1. In problem B, ③ can play on o1 continued by q1 s2 t2 t1.


Problem C. [13]
○ to move first and kill.


Problem D. [14]
○ to move first and get ko.

Differently in problem C., where the status is correct but the first move is wrong. The given first move can be counter-proofed with ❷ on l11, possibly followed by l10 k11 k10 i10. The correct first move is ① on l11 possibly followed by k11 i12 i11 i9 k12 k10 and shortage of liberties for Black. In all three cases an approaching move, pressing in from outside has been overlooked. In problem D. both, the status and first move have to be changed, as ① can kill on b18 possibly followed by d18 b19 e19 c19. If ● would have one more liberty on a16 then the solution given in diagram D. would be correct. In that case there is one more solution leading to a ko, though not favorable for ○ which is ① on b19 possibly followed by b18 c19 e19 e17 d18 a16 a19 a18 pass c18 a19 a18.

References

[1] Dave Dyer, contribution to the computer go email list computer-go@prg.oxford.ac.uk on 13. Oct. 1993.

[2] Dave Dyer, 'Notes on Searches, tree pruning and tree ordering in Go', contribution to the computer go email list computer-go@prg.oxford.ac.uk on 31. May 1994.

[3] According to contributions to the computer-go-email-list, programs devoted especially to life/death have been written by Dave Dyer, Max Golem, Torsten Harling.

[4] P. Aroutcheff, 'Atari en BASIC', Jeux et Stratégie, **31**, 18–19 (1985).

[5] D.B. Benson, B.R. Hilditch and J.D. Starkey, 'Tree Analysis Techniques in Tsumego', IJCAI 79, 50–52.

[6] J. Kraszek, 'Heuristics in the Life and Death Algorithm of a Go-playing Program', CG, 9 (Winter 1988–89), 13–24.

[7] As part of the Japanese 5th Generation Computer Project a computer Go program was developed. This was lead by Noriaki Sanechika, the tsume go part was written by Shinichi Sei.

[8] T. Wolf, 'Tsume go with RisiKo', Proceedings of the Game Festival in Cannes/France, Feb. 1992.

[9] T. Wolf, 'Quality improvements in the tsume go mass production', Proceedings of the Game Festival in Cannes/France, Feb. 1993.

[10] E. Berlekamp and D. Wolfe, 'Mathematical Go Endgames', Ishi Press.

[11] Korean Go journal Baduk, no 8 (1992), p.131.

[12] The error was found by Richard Arundell using `GoTools`. The problem is from Cho Chikun: 'All about life and death', vol 1, p.44, Ishi Press.

[13] Deutsche Go Zeitung, Nr. 7+8/94, p.6, p.39. The error was found by Volkmar Liebscher (Jena/Germany) and checked with `GoTools`.

[14] The error was found by Denis Feldmann (France) using `GoTools` and published in 'L'Aube d'une Ere Nouvelle?', GO, Revue Francaise de GO, no 66, 2d trimestre 1994, pp. 21–24. The problem is from the Ko Dictionary of the Nihon Ki-in.