Forward pruning and other heuristic search techniques in tsume go

Thomas Wolf

Queen Mary & Westfield College, University of London

Abstract

GoTools is the name of a computer program to analyse life & death fights in the game of go. First, a motivation for programing go is given. General aspects of the heuristic search performed in GoTools are discussed. The absolute strength and the effect of forward tree pruning are tested in a competition between GoTools and dan level human players. It is found that with a speed up of a factor of 30 the program still finds for 70% of the problems of a test set the correct first move. A suggestion is made to compare intelligence between contestants (in solving fully enclosed life/death problems) by comparing their relative increase in the time to solve problems with the increase of the difficulty of the problems. The rules of go are explained in the shortest possible form in the appendix.

Introduction

The game of go is a 4500 yr. old Asian board game in which two persons put stones on a board to gain territory and take prisoners. Despite having very few, simple rules and the substantial efforts being spent over the last two decades, go is the most unaccessible of the popular games for computerisation. Projects and incentives include many PhD thesis devoted to the computerisation of go, yearly computer go competitions with prizes of up to 1.6 Mio US$ for a program winning against professional players, and a Japanese National Project in the 1970's which aimed at the development of 5^{th} generation computers and included the computerisation of go by funding a group of researchers full time for a number of years.

In this paper we give an overview of techniques that are used in the heuristic search of a program GoTools that resolves local life and death fights in go. Tactical life & death searches play an important role in the game. Solving local life & death problems (also called tsume go = problem go) is much simpler than playing a full game, which gives computer programs a good chance in comparison with human players. Tsume go is a field in which heuristic search can be successful and in which general principles of heuristic search can be studied and invented. In the section 'Heuristics in GoTools' we try to separate the general ideas which might be useful for the programing of other games from the go specific details by using different fonts.

The program GoTools is the strongest program for medium and large life & death problems in go with a closed boundary. Top amateur players in Japan, England and France have rated it 5th - 6th amateur dan. In 1997 the author received for the development of GoTools the first of the computer go awards, which are given once each year by the Japanese National Computer Go Forum.

Before continuing with the next section in which a motivation for computerizing go is given, it is recommended to the reader unfamiliar with the rules of go to read about them in the appendix A.

Why computer go

Due to the size of the board (about $19 \times 19 = 361$ possible moves at the beginning compared to about 40 moves in chess) and due to the need to look further ahead than in chess (a go game lasts between 250-350 moves compared to about 50 moves in chess) a brute force search like in chess is impossible in go. Whereas in chess a computer has won a match with the world champion already, computer go programs currently have absolutely no chance even against medium strength amateur players, and this despite more than 20 years of research (recently especially in Neural Networks, Genetic Algorithms, Genetic Programming).

This shows that go is a real challenge to program with a potential to force innovation, as so far no single technique has been adequate for it. The typical approach of people in computer science was to use go as a test bed for known techniques. The main result so far was to learn about the weaknesses of these techniques rather than to make progress in computer go. A rare exception of a new successful approach to partial aspects of the game is the work of Conway, Berlekamp, Wolfe and others who developed a mathematical theory for the end game in go [2].
Why is programing go difficult?

- As chains can be arbitrarily large, a move changing the number of liberties of a chain or linking chains or even catching a chain can have global implications. Also in less dramatic situations the influence of a stone can be far reaching. A global causal interdependence is added through the ko-rule which makes the outcome of a local ko-fight depend on whether threats somewhere else on the board are answered or not.

- One can not start with working on elementary building blocks like the investigation of local fights (tactics) and use this knowledge to develop a global picture (strategy). This is because most often local situations are not fully isolated from their surroundings, i.e. the global situation usually matters in local fights.

- Techniques which somehow learn by gathering examples of positions or patterns (databases, neural networks and genetic algorithms where the ability for automatic abstraction is not built in), fail as exclusive tools because subproblems are rarely unrelated to the wider surrounding and the number of medium or larger size positions is too big to be stored in any form (with 361 fields being either empty, white or black an upper bound for the number of possible board positions is 3^{361}). In go it is not possible to simulate intelligence by applying simple algorithms with large speed and memory.

- In the absence of a single general algorithm one has to collect guidance from different tactical and strategic modules which naturally will contradict each other. Without deeper understanding and the ability to "think" (resolving problems by search on a more abstract level) all that can be done is to weight the recommendations of different modules differently and combine them somehow with a failure rate of, say, $> 10\%$. Because the game lasts for typically 250-350 moves and already one blunder can mean the loss, this accuracy is completely unacceptable at a stronger amateur level. In playing a number of these relatively long games there is enough time to find out about weaknesses of the program and to exploit them.

- The above deficiencies are still on a basic level. For a professional playing level a much more abstract and creative thinking would be required.

Life & death searches

If parts of the board are cut off then life & death fights can happen like in the lower left corner of fig. 4 in the appendix A where white moving first could kill black by playing at J. The program GoTools is designed to analyse such positions. Requirements for a life & death program are:

- The program should have different accuracy levels to allow a rigorous analysis as well as a less strict but faster search.

- To guarantee accuracy a branching factor of up to 15-20 might be necessary in order not to miss the single good move. For example, the move on J in fig. 4 appendix A is non-obvious, even for stronger amateur players [3]. The search depth with this problem for GoTools is mostly between 20-40 moves.

- Because life & death problems are small enough to do as many moves as necessary until the position is dead or alive, it opens the possibility to

- do a fast binary $\alpha - \beta$ search, which, for example, means that as soon as a winning move is found, search stops at this level and the loser may try a different move earlier in the sequence of moves,
- learn from the outcome in a terminal node of the search tree, in order to improve the heuristic when back tracing,
- do a depth-first-search for that reason.

- Heuristics must be sufficiently good to safely exclude moves from search (difficult for accurate search which requires nearly everything to be searched), otherwise the depth-first-search can become unboundedly deep. This is the case if one side captures a big bulky chain and does not extend the gained advantage to a win but instead, due to poor heuristics, tries weak moves next and gets itself captured.
- As life & death problems can be very big, good heuristics are essential. To get a very rough estimate, let us assume that no stones are captured, none of the initial n fields in the position becomes forbidden for any side as a consequence of moves done and life or death is recognized when only k empty fields are left. Then full search would involve at most $n \times (n-1) \times (n-2) \times (n-3) \times \ldots \times k$ moves (=nodes of the tree). For finding the correct move after trying on average 50% of the possible moves, the effort would be $\frac{n}{2} \times (n-1) \times \frac{(n-2)}{2} \times (n-3) \times \ldots \times \frac{k}{2}$. If the heuristic to find the best move was perfect and the side moving first could win then full search would still have $(n-1) \times (n-3) \times \ldots \times (k+1)$ moves. That would mean a big speed up of a factor of $\frac{n}{2} \times \frac{(n-2)}{2} \times \ldots \times \frac{k}{2}$, but still a big effort. Heuristics therefore have to be used to cut down as well the tries of the loser. That means that the heuristic should be able to find moves that are guaranteed to be worse than others (e.g. worse than passing) to avoid trying them. Also, if a number of moves are completely equivalent then all but one can be forbidden.
- In a life & death search moves are made until it can be determined whether the position is dead or alive. The sooner this can be decided (quickly algorithmically without trying further moves), the less moves need to be made in total. Having a better algorithm to "see" the status of the position only one move earlier when there are still k moves possible would reduce the total number of moves by *a factor* of k. In the program GoTools much effort has been spent to develop sophisticated life and death recognizing modules (150 kB in Borland Pascal not counting the code for utilities) [4].

Heuristics in GoTools

Depth-first versus breadth-first search

On one hand one wants to do a depth-first-search to learn quickly from each terminal node. On the other hand different branches originating from the same situation may have hugely different sizes unknown beforehand which indicates a breadth-first-search. In GoTools a depth-first-search is done but in each position, as part of the heuristics, all relevant moves are tried and their usefulness is evaluated afterwards including a check of whether the position is already dead or alive. This adds a breadth-first-search component to the otherwise depth-first-search.

Static rules

Most of the heuristic rules are *static* in the sense that they depend only on the place on the board and the other stones on the board. They are of less interest here. A more general static rule is to give moves a low priority which are forbidden for the opponent (they probably could be tried later, "time is money"). Another more general static heuristic is to give moves a higher rating for which the opponent afterwards has fewer legal moves. The branch following such a move is more likely to be smaller and more quickly checked. This strategy is related to the so-called *proof number search* [1].

Dynamic rules

In the little subtree of fig. 1 the numbers give the sequence in which the moves are made and the letter characterizes the field where the stone is placed.
A very simple but useful dynamic heuristic is the following. If, say white, does a move A and black finds that a move B afterwards guarantees blacks win, then white tries B next instead of A (if B is legal for white and not already tried).
Although not quite as useful, other winning moves in the neighbourhood of the node currently under investigation get some credit. In the node '?' in fig. 1 the moves B,F,G would get some credit (D has already been made).

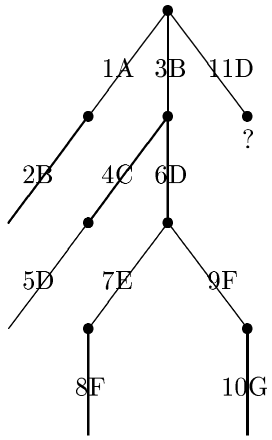


Figure 1. Small subtree

Information flow in the search tree

In traversing the tree, information is gathered and passed on. The main information passed from the root towards the leaves is obviously the position on the board.

This does not only include the stones but also who moves next and the number of ko-threats left for the side which got donated ko-threats (it is the side which in an earlier run lost when no side was donated any ko-threat).

Another type of information passed towards the leaves concerns special situations that might be present. Such extra knowledge is partially used to speed up further searches down the tree.

For example, if the side that wants to live has no space left to built eyes, then its only chance to survive is to catch a chain or to escape and link up with another strong chain of the same color.

Information that flows in the opposite direction, from the leaves towards the root, is the status of each position and the winning move in case it can be won by the side moving next.

Part of the status is information whether the loser lost a ko-fight in the search up to now. That would mean that if in the remainder of the search he would not win in a branch of the tree that includes the lost ko-fight, then the whole search should be repeated with one more ko-threat donated to the loser. (These ko-threats are not really played and answered. The number of times the ko-rule may be violated by one side within one sequence of moves during the search is passed on down the tree.)

Other information propagating towards the root is the number of nodes and the largest depth in the verification subtree. The verification subtree is that part of the tree one gets if the winner plays an existing winning move in each position first, and if the loser tries all moves that seem to have a chance. This part of the tree is enough to verify the win. The number of moves in the longest sequence of the verification tree and its size are used to assess the difficulty of the life & death problem afterwards. Because during the search the winner is not yet known, relevant information (size of all solved sub-branches and longest sequence in them) is stored at each depth and constantly updated.

A hash data base

Whenever a position has been solved during search, then its status and if the status is a win then the winning move are stored in a hash data base. If the hash database gets filled, potentially less useful positions are dropped. To decide that, the length of the longest sequence of moves in the tree solving that position is used. This information therefore has to be stored in the hash database together with the status and the winning move. To get the hash database working properly with the other features of the program (ko, speed up levels) required some effort.

Optimization of heuristic weights

In go heuristic searches with very different branching factors occur.

In so called ladders (a try to catch a two-liberty chain) the branching factor is only 1-2 but the search tree might be as deep as 300. In life & death fights the branching factor can be up to, say, 20 and in full board mid game searches the branching factor can be 200.

The higher the branching factor the more effort should be spend on heuristics. In GoTools, most of the about 30 go specific heuristic rules are rather elementary. More high level information results as a by-product of the life & death test done after each possible move is tried. Some of the rules have an overlap in the situations they cover, while others cover completely different aspects. To get a single heuristic measure, they have to be multiplied with weights and added for each move under consideration. To optimize these weights, a genetic learning algorithm proved to be useful. A population of, say, 20 chromosomes each having 30 numbers included - the weights of the 30 go-specific heuristic rules - undergoes an evolution. As evaluation function, the time for each of the chromosomes to solve a fixed set of life & death problems is taken. Another faster, but not quite adequate evaluation function is to take a set of life & death problems which is not solved, but instead the possible moves are sorted according to their probable usefulness. Depending on which place the unique winning move lands in the sorted list of moves, the chromosome gets a higher or lower credit.

Visualizing the search tree

To facilitate debugging, the program GoTools has the option to step through the tree search move by move, or to proceed with the computation until a specific board position is reached, or a specific node in the tree is reached, or a specific sequence of moves has been made. The ability to go into detail is necessary for debugging, but with search trees containing many thousands of nodes it is also desirable to have a wider perspective of how the tree search proceeds. For that purpose, GoTools has a graphical display which shows the current search depth as a function of time. For each move a short vertical stroke is shown, its color depending on the side making the move. Whenever a terminal node (leaf) of the tree is reached the horizontal value of the next stroke is incremented to the next pixel. This graphical display mode can help to spot strange behaviour, like a sudden drastic increase of the depth of the search or other features. An example: when spotting a short horizontal bar at locally maximal depth the author was reminded that after a winning move in a terminal node was made, the other side will try all possible other previous moves, most of them useless. But because close to a terminal node the situation becomes more obvious the testing of all previous moves can in some cases be short cut and the search be speeded up. The visual appearance of the depth versus node-number graph helped to spot this optimization possibility.

An example when the list of life preventing moves can be reduced: if after a move the position consists of one single chain with two eyes which therefore lives, then the other side can prevent this life only by preventing the move directly by going on to the same field earlier or catching one of the chains that would otherwise live.

Cutting the tree size

The information stored in the hash data base is exact. A further speed up is possible at the expense of accuracy. By disregarding moves from inspection, the search tree gets slimmer and quicker to search. This is also called forward pruning. The aim is to lose as little accuracy as possible with a maximum gain in speed. Simple rules, like giving a strict upper bound to the branching factor (the number of moves to be investigated in each situation) are too restrictive. Instead, it is better to make the branching factor depending on the location within the tree, making it smaller close to the leaves and wider at the root. Another principle is to trust heuristic weights of moves more close to leaves of the search tree as the heuristic is getting more accurate there.

To have different speed up levels, one wants to drop at first the moves which seem most probably useless, and then dropping more and more potentially useful moves if a higher speed up is required. This is done by cutting the tree uniformly, i.e. using many criteria gently instead of using one hard criterion. Some of the requirements are:

- a minimal heuristic weight for a move to be considered initially,
- a minimal heuristic weight for a move to be learned,
- a maximal branching factor without and with learned moves,
- the maximal number of leaves originating from a node,
- the maximal number of branches if one leaf is already found,
- the maximal number of branches if two leaves are already found,
- the maximal difference of the highest heuristic weight of any move in the position,
- the maximal difference of the second highest heuristic weight of any move in the position,
- a maximal search depth (for memory reasons).

With the test in the following section it is possible to demonstrate how appropriate these principles are for a gentle but efficient forward pruning. The aim was to contradict the common belief that the computer's main strength is to apply relatively simple routines very often and fast but as soon as problems get uncalculable difficult the human is unbeatable.

A competition between dan players and GoTools

In the test described below the program and human players are given a set of 64 tsume go problems to solve (shown in appendix B), once very fast and once as accurate as possible. The problems are partially generated by GoTools itself (earlier) and some are taken from [6]. The hardest and most interesting ones have been learned from Denis Feldmann (France). The problems are fully enclosed - the only class of problems GoTools can handle efficiently currently. The aim was not only to compare the absolute speed and accuracy between the program and the players but to compare how much speed is gained and how much accuracy is lost in faster modes for both. Although the program has 5 speed levels, only the fastest (applied in the 'quick mode' in table 1.) and the slowest (applied in the 'slow mode' of table 1.) are used for comparison. In table 1. the total times for solving all problems and the errors in determining the first move and determining the status are shown. The status is the best outcome that the side moving first can enforce. It either is a full success (death if the enclosing side moves first or live if the surrounded side moves first) or a full loss or a ko-situation (when the life or death depends on the outcome of a ko-fight, i.e. on the availability of ko-threats elsewhere on the board). When playing a game the knowledge of the status of a position matters in order to decide whether it is worth to make a move in this position or not.

name	quick mode			slow mode			strength
	time in sec	no of errors		time in sec	no of errors		
		1.move	status		1.move	status	
JB	300	30	-	1500	21	16	4 dan
PW	824	23	15	8737	8	6	5 dan
SC1	927	13	6	1482	5	0	5 dan
SC2	360	30	-	8280	18	12	4 dan
GoTools	15	18	23	484	0	0	?

Table 1: A comparison of the program GoTools with some players each solving 64 problems.

General comments regarding the setup:

- GoTools run in an MSDOS shell under Linux on a 133MHz Pentium. (Already now PC's are available that are 10 times faster.)
- The number in the status column gives how often an error was made when determining whether the position is either a win, ko or loss for the side moving first. A '-' means that no comments were made by the player. The program did not assume anything about the status whereas the players could assume that the problems have a unique best move which gives a straight win or at least a ko.
- In order to find the best move, one does only know in principle which of any two moves is better. This is a relative measure. The status is an absolute measure. If one always knew the best move one would only have to play a single sequence to find the status. In practice both are determined together by searching a tree.
- If the status was found to be a ko then the players usually did not categorize it further. The program determined in which side's favour the ko is and how many external ko threats the other side needs in order to win. This is especially computationally expensive.
- The playing strength of the player is not necessarily equivalent to the person's ability to solve life & death problems, as there is much more to the game than local tactical fights. On the other hand, at the playing level of the contestants one can not afford to be weak in any technique, least of all in life & death fights.

The times to solve individual problems accurately for GoTools and one of the players are compared in figure 2. Each symbol represents one problem with different shapes, depending on whether it was solved correctly by the player or not.

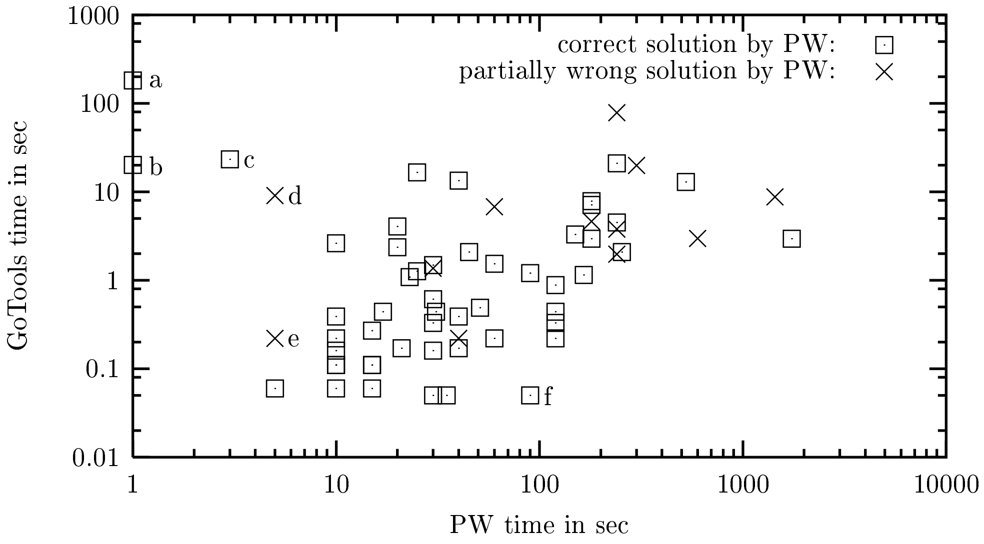


Figure 2. A comparison of times between GoTools and PW. Marks a,b,c,d,e,f refer to the problems with numbers 56,57,17,50,48,38 in Appendix B.

Absolute assessment

Problems close to the vertical axis in figure 2 have been solved faster by the player and problems close to the horizontal axis have been solved faster by the program. The players did normally not make remarks to individual problems but to problems labeled a) and b) PW comments: "This one is a standard shape so it is instantly recognizable". As GoTools does not have a permanent data base, it has no chance here. To problem labeled c) the player remarks: "3 seconds to find the ko ... 30 minutes more to convince myself there wasn't anything better. Still not sure.". Problems labeled d),e) were obviously somehow misjudged by PW. Close to the horizontal axis the problem with label f) has been solved by the program in 50 ms whereas the player needed 90s! It might be interesting for the player to compare his approach with that of the computer.

If one wanted to measure "tsume go intelligence" (or more accurate "fully enclosed tsume go intelligence") of a program or an algorithm then it would not be appropriate to simply compare absolute times for solving some problems. This measure would, for example, be dependent on the computers clock rate. A better measure is to compare the relative increase in time for solving harder problems compared to easier problems. For example, a hard problem is practically unsolvable for weak players whereas strong players can cope relatively better with them. They have a smaller "branching factor". They know that some moves are definitely better than others or they recognize pattern. That means if a weaker player needs a hundred times longer for a hard problem than for an easy problem, then a stronger player may need only ten times as much time for the harder problem than for the simpler one. Looking at figure 2 and disregarding the problems labeled a) and b) (for the above mentioned reason) then we see that both, GoTools and PW, need about 1000 times longer for the hardest problems than for the easiest once. However that comparison is not fair for GoTools. If PW would be given more time for the harder problems in order to get more of them correct (GoTools made no errors) then the relative ratio (time for hard problems)/(time for easy problems) would be in favour of GoTools. Or, expressed differently, if GoTools is allowed to make some mistakes then especially hard problems would profit from the speed up and the above ratio would also become smaller for GoTools than it is for PW in figure 2.

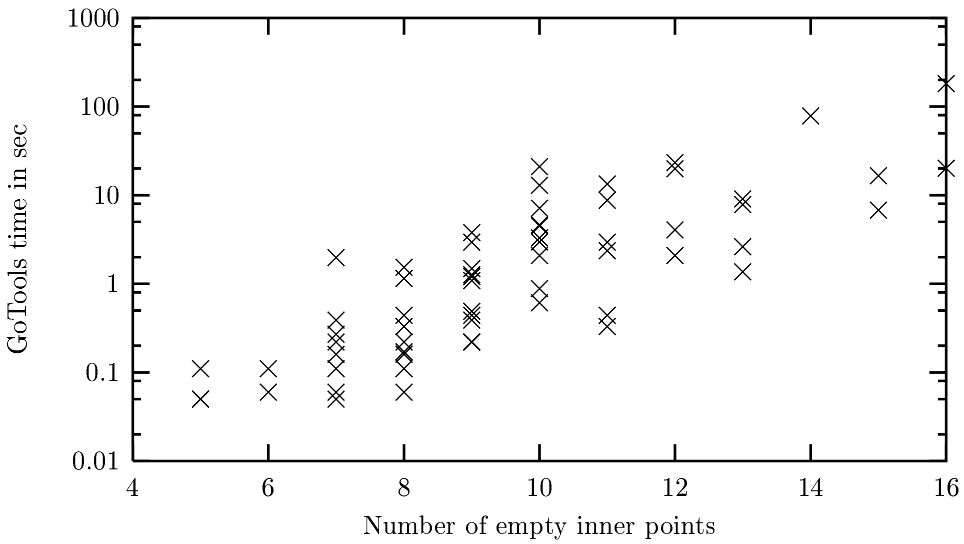


Figure 3. The dependence of run times on the size of problems

In figure 3 the time needed by GoTools to solve problems accurately is plotted in dependence of the number of empty inner points of the problems. We see that in average the time grows exponentially with the size of the problem but we also see that times may vary by a factor of up to 40 for problems with the same number of empty inner points. This is partially a consequence of ko-investigations taking much longer if they are necessary. Also, if weak stones inside the position are captured then new empty points appear.

To mention it again, fully enclosed life/death fights are rather rare in go games, especially in games between better players. It therefore must be said that the program GoTools in its current form is useful as a stand-alone program to analyse difficult situations (appropriately closed if possible) and as a training tool but not as part of a full game playing program.

The effect of forward pruning

With the following comments we will try to assess the pruning of the search tree in GoTools as discussed in a section above.

- Even in its most inaccurate mode when the program is, for example, 100 times faster than J.B. solving the problems slowly, it still finds more accurate first moves!
- To summarize, the test shows that forward pruning as done in GoTools is able to keep an accuracy of $> 70\%$ of finding the correct first move and still gaining speed with a factor > 30. In the fastest mode the program spent in average only 234 ms for each problem, allowing it to search trees with an average of only 9.6 terminal nodes! This result is to be valued even higher in the light that tsume go is hard and resistant to any pruning as discussed above.

WWW

The program can solve problems but also invent new problems. 40.000 computer generated life & death problems are freely available. They have been used as a training set by a number of people for their GA and NN programs. Information of how to get the problems, the program including references is given in `www.qmw.ac.uk/~ugah006/gotools`. Each day two problems are posed in `www.qmw.ac.uk/~ugah006/tsumego`.

Acknowledgement

The author thanks Jonathan Buss, Stanley + Stella Chang and Philip Waldron for taking part in the tsume go test. Mike Waite is thanked for comments on the manuscript.

Appendix A: The rules of go

Go is a 4500 yr. old Asian board game in which two persons alternatively place plain, equally shaped stones, one white the other black, on the intersections of a 19×19 grid (see fig. 4).All neighbouring stones of one color make up a so-called chain. For example, Ⓐ Ⓑ Ⓒ are a chain, Ⓓ is a chain and 🅐 🅑 are a chain. The main property of a chain is its number of empty neighbouring fields, called liberties, for example the chain with ⒶⒷⒸ has the 5 liberties A,B,C,D,E. Any chain needs at least one liberty, otherwise it is captured and taken off the board. If white would play next on F then 🅐 🅑 would be captured, taken off the board and white would have got two prisoners. Naturally it is forbidden to make a suicidal move. Therefore white is not allowed to go on G but black is allowed to go on G because this black stone would capture white and then have 3 liberties. Differently in the right upper corner of the board. Here black can neither go on H nor on I because in each case white would have one liberty left and therefore not be captured, hence the black move would be suicidal and is therefore forbidden.

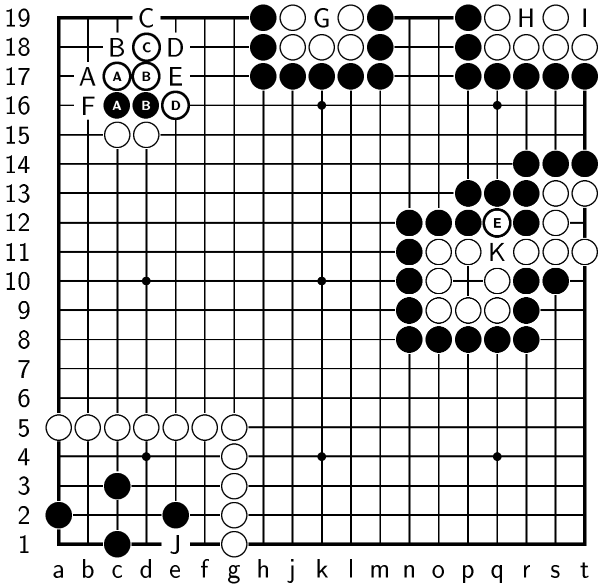
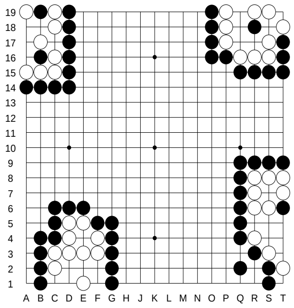


Figure 4. Go board with stones. Letters only mark fields and stones which are referred to in the text.

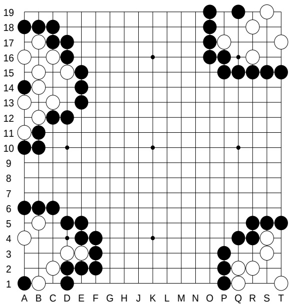
The reason that the right position can not be captured (→ is alive) is that it has *two* liberties only surrounded by white - so called eyes. The aim of the game is to surround as many fields as possible by living chains of ones color and to capture as many opponent stones as possible prisoners. The game naturally ends when for each chain it is clear that it either lives or that it is dead, i.e. that it can not escape capture, and there is no more space on the board left to start building own living chains, to increase own territory, or to diminish territory of the opponent. A final rule forbids the recapture of an opposite single stone immediately if the same position would result. For example, if on the right edge of the board in fig. 4 white would play at K then white would have two eyes and live. However, if black is to move next, then black would go on K and capture the single white stone Ⓔ. After capture, the single black stone has only one liberty and could be recaptured, but the ko-rule forbids that so white has to make another move. If black then links its single black stone to the other black stones by playing where Ⓔ is now then all the white stones in this local position will die. Therefore white's aim will be to find a threat elsewhere on the board which black must answer and then to recapture the single black stone. This search of threats and recaptures goes on until one side has not got a threat which is at least as important as the whole position around this ko and then the other side wins this ko-fight. This so-called ko-rule has severe implications for any computer go program that aims to be accurate.

To enable someone to play go, the above rules would have to be illustrated with many more examples, which are omitted here.

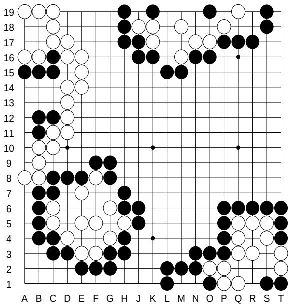
Appendix B: The 64 problems of the tsume go test



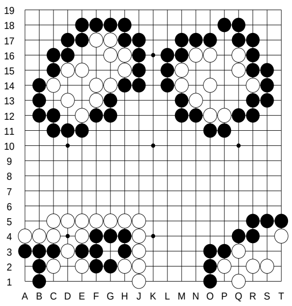
Lower **L**eft: 1: Black to move first
Lower **R**ight: 2: Black to move first
Upper **R**ight: 3: Black to move first
Upper **L**eft: 4: Black to move first



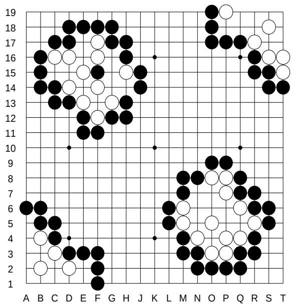
Lower **L**eft: 5: White to move first
Lower **R**ight: 6: Black to move first
Upper **R**ight: 7: White to move first
Upper **L**eft: 8: Black to move first



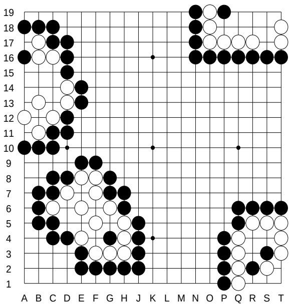
Lower **L**eft: 9: White to move first
Lower **R**ight: 10: Black to move first
Upper **R**ight: 11: Black to move first
Upper **L**eft: 12: Black to move first



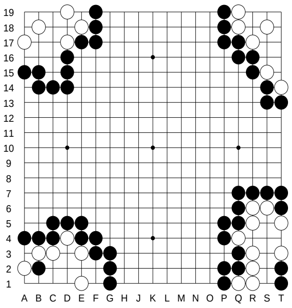
Lower **L**eft: 13: Black to move first
Lower **R**ight: 14: White to move first
Upper **R**ight: 15: White to move first
Upper **L**eft: 16: Black to move first



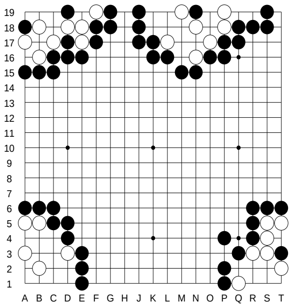
Lower **L**eft: 17: Black to move first
Lower **R**ight: 18: White to move first
Upper **R**ight: 19: White to move first
Upper **L**eft: 20: White to move first



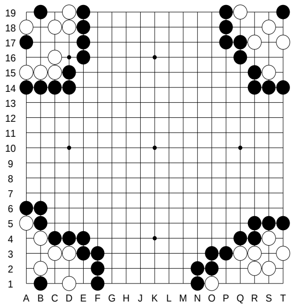
Lower **L**eft: 21: Black to move first
Lower **R**ight: 22: Black to move first
Upper **R**ight: 23: Black to move first
Upper **L**eft: 24: White to move first



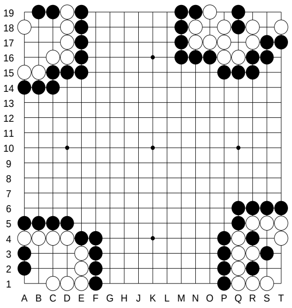
Lower **L**eft: 25: White to move first
Lower **R**ight: 26: Black to move first
Upper **R**ight: 27: White to move first
Upper **L**eft: 28: White to move first



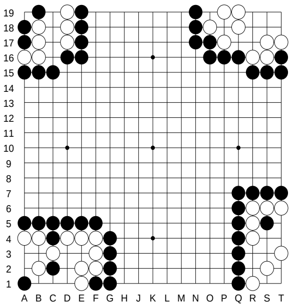
Lower **L**eft: 29: White to move first
Lower **R**ight: 30: Black to move first
Upper **R**ight: 31: White to move first
Upper **L**eft: 32: White to move first



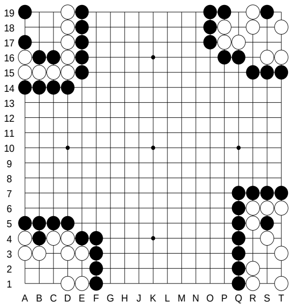
Lower **L**eft: 33: White to move first
Lower **R**ight: 34: Black to move first
Upper **R**ight: 35: White to move first
Upper **L**eft: 36: Black to move first



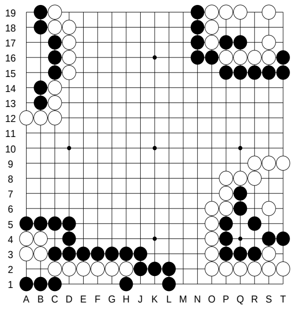
Lower **L**eft: 37: Black to move first
Lower **R**ight: 38: Black to move first
Upper **R**ight: 39: Black to move first
Upper **L**eft: 40: Black to move first



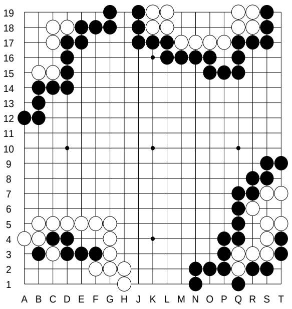
Lower **L**eft: 41: Black to move first
Lower **R**ight: 42: Black to move first
Upper **R**ight: 43: Black to move first
Upper **L**eft: 44: Black to move first



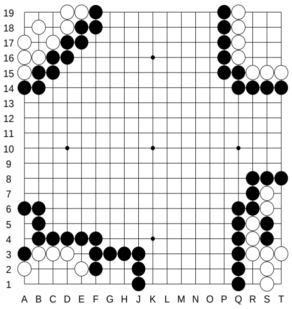
Lower **L**eft: 45: Black to move first
Lower **R**ight: 46: Black to move first
Upper **R**ight: 47: Black to move first
Upper **L**eft: 48: Black to move first



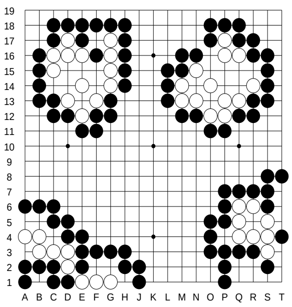
Lower **L**eft: 49: Black to move first
Lower **R**ight: 50: Black to move first
Upper **R**ight: 51: Black to move first
Upper **L**eft: 52: White to move first



Lower **L**eft: 53: Black to move first
Lower **R**ight: 54: White to move first
Upper **R**ight: 55: White to move first
Upper **L**eft: 56: Black to move first



Lower **L**eft: 57: Black to move first
Lower **R**ight: 58: White to move first
Upper **R**ight: 59: Black to move first
Upper **L**eft: 60: Black to move first



Lower **L**eft: 61: Black to move first
Lower **R**ight: 62: White to move first
Upper **R**ight: 63: Black to move first
Upper **L**eft: 64: White to move first

Instructions given out with the test

A collection of 64 problems is to be solved twice, the first time very quickly, taking only a few seconds for each problem, just enough to make a sensible guess. The total time for the 64 problems is to be recorded. For each problem mark the best move with eg. a pen in the diagram and if you can, note what the status is (l for life, d for death, k for ko). When going through the problems a second time one should try really hard and note for each problem how long it took to solve it, or how much time one has spent on it, also marking the first move and recording the status on the sheets.

Comments to the instructions

One could have specified a strict time limit, like 10 min for the quick mode and 5 hours for the slow mode. Fixed time limits would be useful for comparisons between humans. If a concrete time would be given in this test then it would be open to the criticism that the time chosen would just suit GoTools and that would not be fair for the player. One can not expect from a player to improve with every little more time he/she gets. Every player has their own pace of solving go problems. So it is more fair to the player to have no strict time limits, only guidelines and to take the time that has been used into consideration when counting the errors.

Solutions of the 64 problems

The status of the problems is usually a straight win (death if the attacker moves first, live if the defender moves first). If the outcome is ko (a local ko fight develops with the outcome depending on the availability of outside ko-threats) then a (K) is printed behind the best move. For some problems (8,17,19,20,21,27) the winning move may only be better than another move because the ko it generates is better for the side moving first than the ko generated by the second best move.

1. D1	**2.** S5	**3.** Q17	**4.** C17	**5.** B3	**6.** S1
7. T18	**8.** A12 (K)	**9.** D6	**10.** R2	**11.** N19 (K)	**12.** A13
13. D1	**14.** S1	**15.** P13	**16.** F16	**17.** A4 (K)	**18.** N6
19. R19 (K)	**20.** G15 (K)	**21.** E5 (K)	**22.** S1	**23.** S17	**24.** C14
25. B1	**26.** S4	**27.** S16 (K)	**28.** C17	**29.** B4	**30.** R2
31. M17 (K)	**32.** C19	**33.** A3 (K)	**34.** T4 (K)	**35.** S16 (K)	**36.** B17
37. C2	**38.** T2	**39.** R19	**40.** B17	**41.** D3	**42.** S4
43. S18	**44.** C17	**45.** B2	**46.** S2	**47.** S17	**48.** C18
49. F1 (K)	**50.** S7	**51.** S18 (K)	**52.** A16 (K)	**53.** A2	**54.** S1
55. N19	**56.** A15	**57.** B1	**58.** T6	**59.** S17 (K)	**60.** B19 (K)
61. B1	**62.** R2	**63.** O15 (K)	**64.** F15 (K).		

References

[1] Allis, L.V. *Searching for solutions in games and Artificial Intelligence* M.Sc. thesis, ISBN 90-9007488-0.

[2] Berlekamp, E. and Wolfe, D., *Mathematical Go Endgames*, Ishi Press International (1994).

[3] Wolf, T. *The Diamond*, British Go Journal, 108 (1997), 34-36.

[4] Wolf, T., *The program GoTools and its computer-generated tsume go database*, Proceedings of the Game Programming Workshop in Japan'94, Hakone, p 84-96, 1994.

[5] Denis Feldmann (France), private communication.

[6] Deutsche Go Zeitung, Nr. 3, 1988.