Division of Informatics, University of Edinburgh

Centre for Intelligent Systems and their Applications

Applying adversarial planning techniques to Go

by

Steven Willmott, Julian Richardson, Alan Bundy, John Levine

Informatics Research Report EDI-INF-RR-0103

Division of Informatics **February 2001**

http://www.informatics.ed.ac.uk/

Applying adversarial planning techniques to Go

Steven Willmott, Julian Richardson, Alan Bundy, John Levine

Informatics Research Report EDI-INF-RR-0103

DIVISION *of* INFORMATICS
Centre for Intelligent Systems and their Applications
February 2001

appears in Theoretical Computer Science, Vol. 252, No. 1-2, February 2001, Elsevier, 45–82

Abstract :
Approaches to computer game playing based on alpha-beta search of the tree of possible move sequences combined with a position evaluation function have been successful for many games, notably Chess. Such approaches are less successful for games with large search spaces and complex positions, such as Go, and we are led to seek alternatives. One such alternative is to model the goals of the players, and their strategies for achieving these goals. This approach means searching the space of possible goal expansions, typically much smaller than the space of move sequences. Previous attempts to apply these techniques to Go have been unable to provide results for anything other than a high strategic level or very open game positions. In this paper we describe how adversarial hierarchical task network planning can provide a framework for goal-directed game playing in Go which is also applicable both strategic and tactical problems.

Keywords : planning, computer go

Theoretical Computer Science 252 (2001) 45–82

Theoretical Computer Science

www.elsevier.com/locate/tcs

Applying adversarial planning techniques to Go

Steven Willmott[a, *], Julian Richardson[b], Alan Bundy[b], John Levine[c]

[a]*Laboratoire d'Intelligence Artificielle, École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland*

[b]*Institute of Representation and Reasoning, Division of Informatics, University of Edinburgh, Scotland, UK*

[c]*Artificial Intelligence Applications Institute, Division of Informatics, University of Edinburgh, Scotland, UK*

Abstract

Approaches to computer game playing based on α–β search of the tree of possible move sequences combined with a position evaluation function have been successful for many games, notably Chess. Such approaches are less successful for games with large search spaces and complex positions, such as Go, and we are led to seek alternatives. One such alternative is to model the goals of the players, and their strategies for achieving these goals. This approach means searching the space of possible goal expansions, typically much smaller than the space of move sequences. Previous attempts to apply these techniques to Go have been unable to provide results for anything other than a high strategic level or very open game positions. In this paper we describe how adversarial hierarchical task network planning can provide a framework for goal-directed game playing in Go which is also applicable both strategic and tactical problems. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Computer games; Go; Planning; Adversarial planning

1. Introduction

Most approaches to computer game playing are based on game tree search and position evaluation functions (*data-driven* approaches). Data-driven approaches are appropriate for games with low branching factors, and for which it is possible to accurately assign values to positions which indicate who is winning. While this approach has been very successful for many games including Chess, it has been less successful when

* Corresponding author.
E-mail addresses: willmott@lia.di.epfl.ch (S. Willmott), julianr@dai.ed.ac.uk (J. Richardson), bundy@dai.ed.ac.uk (A. Bundy), johnl@aiai.ed.ac.uk (J. Levine)

0304-3975/01/$ - see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S0304-3975(00)00076-1

applied to games with high branching factors and complex positions, such as Go or for games with a high degree of uncertainty such as Bridge.

An alternative to the data-driven approach is *goal-driven* search, in which a single agent searches for a way to satisfy its goals in the game. Goal-driven search has been extensively explored in the Artificial Intelligence literature, in particular as hierarchical task network (HTN) planning [42, 15]. When multiple agents need to be modelled and can compete against one another, this approach becomes *adversarial planning*. In this paper we describe how adversarial hierarchical task network planning can provide a framework for goal-directed game playing in Go. We consider different types of goal expansion in HTN planning and suggest how to make planning approaches more applicable to tactical problems.

1.1. Paper overview

In Section 1.2, we review goal-driven and data-driven approaches to making move choices in game playing. Section 1.3 presents an overview of previous work on computer Go (Section 1.3.2) and Go's search space (Section 1.3.1). Section 1.4 completes the first section with an outline of previous work in the field of adversarial planning. The main body of the paper includes the following two principal contributions:

- The description of a new adversarial planning framework (Section 2.2), based on hierarchical task network planning (Section 2.1). The framework models two agents, each of which attempts to satisfy its own goals while refuting those of its opponent. This behaviour is achieved by backtracking, and the enforcement of a time linearisation during goal decomposition (discussion of the decomposition strategy can be found in Section 2.3).
- The application of the planning framework to the game of Go (Section 3). In order to prove the concept without coding large amounts of knowledge, our implementation, GOBI, focuses on life-and-death problems of the type found in Go teaching books (although there is no restriction to *enclosed* problems).

GOBI was systematically tested on examples from volume I of "Graded Go Problems for Beginners" [49], finding the correct answer for 74% of the problems. We analyse both successes and failures in Section 4.4.

The remainder of the paper examines the advantages and disadvantages of applying a goal-driven approach to computer Go (Section 5), and compares our work to adversarial planning systems in other domains (Section 5.4).

1.2. Goal-driven and data-driven approaches

Within a specific game, move or action choices often depend upon the state of the game, the phase of the game (e.g. opening, endgame etc.), the future actions of the opponent, the ability of a player to follow up an action appropriately and many other diverse factors. It is these interacting influences on the choice and effect of moves which make games so fascinating for human players and so challenging for machines.

In computer game playing there are two main approaches to making move choices:

- *Data-driven*: At each step, rules, patterns or heuristics are applied to the game state to suggest useful moves. The resulting set of plausible actions is then evaluated using search in the tree of moves. Each move is played out in a world model followed by the possible responses of the opponent. The search continues until the leaves of the tree are reached. [1] These leaf nodes are then evaluated and used to select one of the original plausible actions as the one which leads to the most desirable (by some measure) set of leaf states.
- *Goal-driven*: During play, a goal-driven system keeps a number of abstract goals [2] in an agenda. The goals in the agenda represent the things the system would like to achieve in the short, medium and long term. To choose a move, goals are expanded into plans (which are conjunctions of goals at lower levels of abstraction) and eventually into concrete moves (actions in the world). Repeated decompositions form a plan for achieving the goal.

In a data-driven search tree, each node represents a possible game position and has one branch for every move suggested in that position. In contrast, each node in a goal-driven search tree represents a *plan* for achieving the top-level goal with some parts still sketchy (abstract goals) and others fixed (concrete actions), and each node in the search tree has one branch for each way the system suggests to further refine the plan.

Which approach (goal-driven or data-driven) is most advantageous is heavily dependent upon the domain, in particular on the size of the data-driven and goal-driven search trees. In Bridge, for example, the locations of the cards are not in general known during play, which leads to a large space of possible card plays and therefore a prohibitively large data-driven search tree. [3] Smith et al. (in [38]) show that a goal-driven approach can very successfully play Bridge, and work described in [18] demonstrates that a relatively small number of operators is sufficient to describe all the relevant plays.

1.3. The game of Go

The game of Go is considered by many to be the next great challenge for computational game playing systems. It presents new, significant and different challenges to Chess which has been long been considered the “task par excellence” for Artificial Intelligence and computer game playing [3]. A good introduction to the game can be found in [4].

[1] Which nodes are the “leaves” can be variously defined by a depth cut-off point, quiescence, or further domain-dependent heuristics.

[2] Abstract goals are aims which cannot in general be achieved by a single primitive action in the world.

[3] A data-driven search can still be practicable, but only by dramatically limiting the part of the move space which is searched. For example, Ginsberg’s GIB [20] adopts a sampling approach, aiming to sample a representative part of the search tree, and choosing its moves on this basis.

1.3.1. Size of the Go search space

The search space of Go is both wider and deeper than that of Chess; there are estimated to be about 10^{170} states (cf. Chess $\approx 10^{50}$), games last approximately 300 moves (cf. Chess ≈ 80) and the branching factor at each turn is on average 235 states (cf. Chess ≈ 35). The number of possible games of Go is estimated at 10^{700} compared to $\approx 10^{120}$ for chess [5]. It is often hard to evaluate the relative strength of Go positions during play, since guiding principles such as value of material which are very useful in chess are often misleading for Go. The difficulty of Go is as much due to this difficulty in state evaluation as to the large raw search space. Useful complexity results have also been shown:

- Ref. [24] shows that determining the eventual winner for an arbitrary Go position on an $n \times n$ board is PSPACE-hard. [4]
- Ref. [33] extends this result to show that deciding whether or not black can win from an arbitrary position is EXPTIME-complete (taking into account situations arising form Ko positions).

Given these difficulties, the brute-force game tree search which has been so effective for Chess will potentially have much greater difficulty with Go.

1.3.2. Approaches to computer Go

Although Go has received far less attention than Chess in terms of research, there have been many varied approaches to computer Go:

- Hybrid approaches such as Go 4++ [6], Many of Faces of Go [16] and Handtalk [6] are by far the most successful at playing the complete game to date. These systems are mainly data-driven (Go 4++ for example, works almost exclusively on pattern recognition) but their long period of development (10–15 years for Many Faces of Go) has seen the addition of many other types of reasoning and specialist modules.
- Non-symbolic techniques have been used to learn/evolve controllers and rules based upon patterns of stones for use during play. The techniques applied include genetic programming [10], genetic algorithms [32, 22, 12], and neural networks [13]. These approaches have so far been less successful than the hybrid programs but have the advantage that Go knowledge does not need to be added by hand.
- Cazenave's GoGol [8] applies learning techniques to good effect. An off-line program uses introspection to prove theorems about Go tactics which can be used to generate knowledge useful in pruning Go search trees. This knowledge is then added into GoGol's playing procedures. GoGol was the top ranked non-commercial program [9], finishing 6th out of 40 participants in the IJCAI'97 international computer Go tournament [17].
- Progress has also been made by focusing on specific subproblems in Go. Wolf's GoTools [47, 48] uses very deep search to analyse closed "life and death"

[4] Ref. [41] gives proofs for a similar result for a class of generalisations of Chess to $n \times n$ boards.

game positions and can often solve very complex problems. [26] applies combinatorial game theory to endgame situations, which enables precise calculation of the values of sub games and hence perfect play, but is intractable except for the last few moves of the endgame.
- There have also been several applications of planning techniques to Go. These systems are discussed separately in Section 1.4.

The top Go programs are now able to challenge good amateur players, but there remains substantial room for improvement. The advantages mentioned in Section 5.1, earlier work on Go planners [37, 23], and the success of the goal-driven approach in other domains (notably Bridge [40]) suggest that a goal-driven approach may be useful. It also has much psychological validity, since protocol analysis indicates that Go players consider few candidate moves, and concentrate on their own and on their opponents' purposes [36]. Finally, even in data-driven approaches to computer Go, it is still necessary to consider high-level goals, for example in order to decide whether or not a satisfactory result has been achieved in life and death problems (e.g. some strings may be allowed to die but others must live) [48].

1.4. Applications of adversarial planning

Domains in which agents compete against one another are termed *adversarial*. The first attempts to use goal-driven reasoning in adversarial domains include work by Pitrat [29] and Carbonell [7]. The former was extended by Wilkins in [44] to produce the PARADISE system for Chess. Using goals to guide search, PARADISE was able to find moves which involved looking as far ahead as 20 ply in certain situations, a depth well beyond its contemporary search-intensive competitors. More recent work studies battlefield management [1], command and control [50], and Bridge [39, 40]. The work on Bridge is perhaps the most successful application of goal-driven reasoning to games to date, presenting a system for Bridge declarer play (TIGNUM2) good enough to beat the current top commercial computer player [39]. Frank et al. [19] describes the FINESSE program for Bridge which also applies a plan-based approach. Extensive testing has shown that FINESSE can find optimal plans for single-suit play, and correctly calculate their probability of success.

Go has been used as an application domain for adversarial planning systems:
- The INTERIM.2 Go program [30] was probably the first Go program to use the idea of goals to guide move choice.
- This initial interest was followed by two Go planners due to Sander and Davies [37] and Lehner [23] which both addressed only very open positions.
- More recently, Hu [21] also concentrates on high-level strategy looking at the possible use of multipurpose goals.
- Finally, the GOBELIN system developed by Ricaud [31] also has aspects of a goal-driven approach. GOBELIN uses an abstract representation of the Go game state to form plans before mapping these back into the ground state.

A common feature of all[5] of the goal-driven Go programs to date is that although some are able to report good results for high-level strategic planning [37, 23] or during opening play [31], none adequately addresses the tactical level of Go play. There has been little success for this type of approach in the middle or end-game where tactical and strategic considerations become closely linked. Contributions of this paper include offering an explanation for this lack of progress, based upon the type of decomposition strategy used during planning, and presenting an approach which makes goal-driven reasoning applicable at both the strategic and tactical levels.

2. An adversarial planning architecture

This section describes an adversarial planning architecture which models goal-driven reasoning for adversarial domains.[6] The goal-driven approach and use of abstract plans is motivated by work on hierarchical task network (HTN) planning. HTN systems were first used in Noah [34] and Interplan [42] and have since been extensively studied in the AI planning field. Erol et al. [15] give a complete definition for an HTN scheme and present Ucmp, which is a provably sound and complete HTN planner and provides a good template for this type of system.

2.1. Principles of HTN planning

HTN planning is based on three types of object: *goals*, *operators* and *plan schemas*. Operators are actions which can be performed in the world (such as flicking a switch, taking a step). Goals are more abstract and express aims in the world such as “Go to the Moon”, “Become Prime Minister”. Schemas (also called task networks or methods), specify the subgoals which must be achieved in order to satisfy the goal. For example, the following schema expresses the fact that G can be achieved by satisfying the conjunction of subgoals G_1, G_2 and G_3:

$$G \Rightarrow G_1 + G_2 + G_3.$$

The G_i should be at a lower level of abstraction than G, and can generally be satisfied in any order. Operators are at the lowest level of abstraction.

Given these three types of object, HTN planning starts with an initial world state and a set of goals which form the initial abstract plan. The plan is then refined step by step by expanding the goals within it. Goals are expanded by selecting a schema whose antecedent (the G above) matches the chosen goal, and replacing the instance of G in the current plan by the subgoals (the G_i above) listed in the consequent of the schema.[7] As the planning process continues, interactions, incompatibilities and

[5] Except Interim.2 which applies data-driven search for local tactical lookahead.

[6] More details (including a precise breakdown of the planning algorithm) can be found in [45].

[7] Some planners may also instantiate variables as part of this expansion process which adds extra complexity [14].

conflicts may arise between combinations of goals. These “interactions” in the plan must be resolved, which can result in backtracking and (in partial order planners) ordering constraints between goals.

The process is complete when all goals have been expanded into sets of operators and all arising interactions have been resolved. The sequence of operators thereby generated should, upon execution in the initial world state, lead to the achievement of the planner’s goals in the world.

The characteristics of an HTN planner are principally determined by its goal expansion strategy. At any stage a plan can be *totally ordered*, which means that every step is time ordered with respect to every other step or *partially ordered*, in which case some steps do not have ordering constraints between them. A partial order planner allows goals to remain unordered until ordering becomes necessary whereas a total order planner chooses a linearisation over the plan steps to ensure that the steps are always totally ordered. Planners may also use two different inference mechanisms: backward chaining and forward chaining. Backward chaining planners begin with the goal state and use their knowledge to work backwards towards the start state, forward chaining planners do the opposite – starting with the initial state and applying sequences of operators to reach the goal.[8] Most standard HTN planners in use are backward chaining partial order planners.[9]

The extension of this idea into adversarial domains is non-trivial since plans are no longer sequences of actions but trees of contingencies which take into account the actions of opponents. The interactions in the plan are considerably more complex and serious since the goals of opponents in the world are often conflicting and the planning agents are non-cooperative. HTN planning for adversarial domains is therefore computationally considerably more complex than HTN planning in standard domains.

2.2. Adversarial planning framework

The adversarial planner presented in this paper models two agents (named Alpha and Beta) which represent two players (*adversaries*) in a game.[10] To solve a problem in the domain, each agent is given a set of input goals to achieve. We assume that the input goals of the two agents are mutually exclusive, so that a successful plan for one agent to satisfy its input goals acts as a refutation of the other agent’s input goals.[11] Alpha and Beta’s tasks are to achieve their own goals while preventing the other agent from achieving theirs.

[8] There is a distinction between *forward* and *backward chaining* and *data* and *goal driven*: forward and backward chaining describe ways of traversing the search space, whereas data and goal driven relate to the motivations behind move choice.

[9] In terminology we follow [25].

[10] The framework can be generalised to include more than two players.

[11] We could usefully relax the restriction that the input goals of the two agents be mutually exclusive, by instead requiring only that when one agent satisfies its input goals, the other agent’s goals become less desirable.

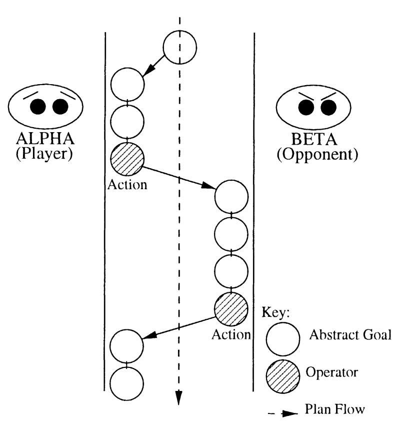

Fig. 1. Planning steps alternating between two agents.

Each agent keeps an open agenda of goals (descended from the input goals by expansion steps) which represents its current plan of action. Using these plans the agents attempt to find sequences of moves which satisfy their input goals. As the agents take turns to try and achieve their goals, their interaction in the world can be used to find a plan for the problem.

Both agents (Alpha and Beta) share the same knowledge base of plan schemas. For any given problem the two agents are likely to use different parts of the knowledge base (one using attacking plans, the other using defensive plans for example), although overlaps are not excluded (some attacks are powerful defences). The system allows the two agents to take control of the reasoning apparatus in turns (Alpha takes control first). Once an agent has control, it expands some of its abstract goals until it is able to decide upon a concrete action. The chosen action is then performed in a world model [12] before control is passed to the other agent. Fig. 1 shows the flow of control during the reasoning process. An agent may need to expand several abstract goals before being able to decide upon an action in the world. During this "active" period, it uses its own agenda of goals and has control of the shared reasoning apparatus. Once an action is chosen, control passes to the other agent.

At any one time an agent has a plan which consists of actions already taken (square boxes in Fig. 2) and goals at various levels of abstraction (circles in Fig. 2). The actions (squares) are represented in the world model, the abstract goals (circles) are held in the agenda.

A planning step involves selecting an abstract goal (such as X in Fig. 2) and expanding it. A plan schema is selected for X which expresses how X could be achieved using a conjunction of subgoals at a lower level of abstraction. For example, in Fig. 2, X is

[12] A world model is not a standard feature of HTN planners – see Sections 2.3.2 and 5.4.3 for more explanation of its use.

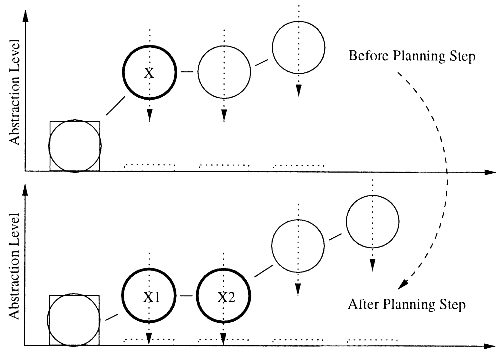

Fig. 2. Plan refinement: abstract goals are expanded to and replaced by sets of subgoals at lower levels of abstraction.

replaced in the plan by the two subgoals $X1$ and $X2$. Once expansion has reached the lowest level of abstract goals, these lowest level goals need to be shown to be already true or replaced by actions which make them true.

Once one of the agents (Alpha say) has achieved all of its goals (been able to perform actions in the world model which make them true), it knows that it must have satisfied its top level goals (since all its subgoals are strictly descended from these). The opposing agent is made aware of this fact and may force backtracking.[13] Backtracking is only forced by an agent when the other agent achieves *all of its input goals*. Thus, one agent can allow the other to achieve any number of subgoals while pursuing its own agenda. This trading of subgoal achievement ("while my opponent is achieving X, I will achieve Y which seems more important to me") is a vital part of many games. Agents are allowed to backtrack to any of their previous goal or expansion choices but only to their *own* decisions. Neither agent may force the other to change plans directly.

The backtracking activity explores the various interacting plans Alpha and Beta have for the situation and creates a plan tree as shown on the left of Fig. 3. Each choice made by an agent creates a new branch. Underlying the plan tree is the *contingency tree* which is found by removing all the abstract goal decomposition steps in the plan tree to leave only the operators/actions (shown on the right in Fig. 3). Moves in the contingency tree are *directly descended* from the goals of the two agents, and the tree structure naturally reflects the interactions between the two adversaries. Taking any branch, the moves chosen near the leaf (at the end of a move sequence) descend from the same goal as those near the root of the tree, and therefore serve the same purpose.

[13] This rule expresses the fact that Alpha and Beta's goals are in some sense mutually exclusive (and the agents must prevent each other from satisfying their respective goals).

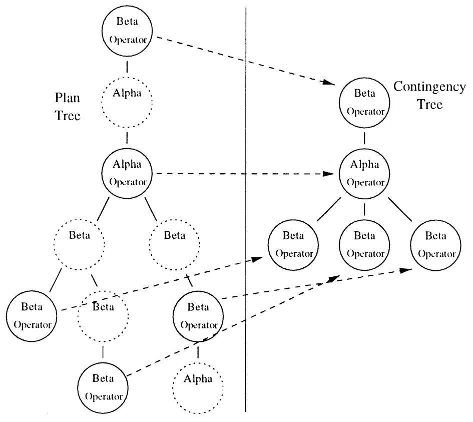

Fig. 3. The plan tree on the left is reduced to the contingency tree on the right by dropping the abstract reasoning nodes.

Since all search is based upon the knowledge available, the knowledge base can be used to control the extent (time taken) of a particular search. Restricting the amount of knowledge (plan schemas) taken into account by the planner leads to less complete plans/counterplans but reduces the amount of time taken during search.

The final contingency tree acts as a form of proof that the first move is a good step towards achieving Alpha's goals. Hence, it supports the choice of the first move in the tree. [14] In general, the final contingency tree contains only a small subset of the move tree which would be generated by considering all the available options at each turn as in a data-driven approach (see Section 5.1).

2.3. *Goal decomposition schemes and a world model*

To better describe how the planner works, this section concentrates on its goal decomposition scheme. The method of goal decomposition defines how the planner applies its knowledge to problems and defines the fundamental characteristics of the planner.

2.3.1. *Standard HTN goal decomposition*

The standard scheme for HTN decomposition (widely used to good effect in non-adversarial domains) is as shown in Algorithm I. [15]

At each refinement step a goal is chosen and decomposed to reduce the overall level of abstraction in the plan. In general, all goals are kept at roughly equal levels of

[14] The tree itself can also be used to respond to any of the opponent's moves which are represented in it, but replanning may be required if other moves are made. The question of how long (for how many moves) such a plan remains valid is not addressed here.

[15] This algorithm is a simplified version; for more detailed coverage, see [15].

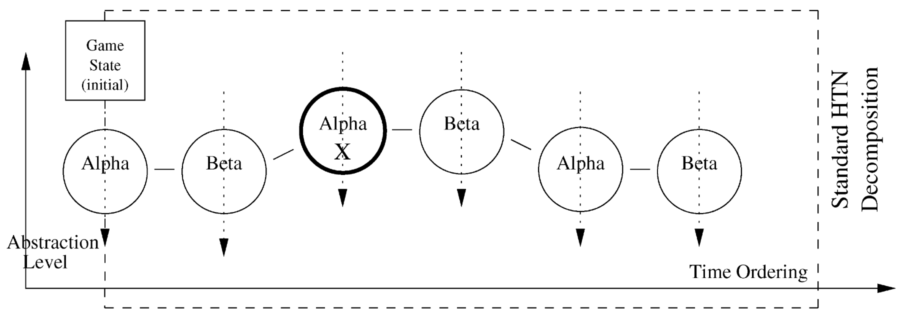

Fig. 4. Goal decomposition in "standard" HTN planning. The dashed arrows represent the decomposition of goals (eventually) into concrete actions.

abstraction. The average abstraction level of the goals in the plan

```
procedure Refine-Plan
    while Not Empty(Agenda) do
        1. choose a goal NextGoal from Agenda to decompose
        for goal NextGoal do
            2. select a decomposition schema S
            3. remove goal NextGoal from Agenda
            4. insert schema S into Agenda
```

Algorithm 1. At each step one goal is chosen and decomposed. (*Agenda* is the list of open goals held by each agent.)

then gradually decreases as the goals are decomposed, eventually reaching the state where all goals are concrete actions in the world (primitives) and the plan can be executed. This approach is a purely backward chaining strategy in the space of plans. The system starts with the top-level goal state and systematically fleshes out the plan using plan knowledge. The algorithm does not specify any particular subgoal ordering, and therefore is suitable for both total and partial planning. A total order planner would additionally make execution order choices in step 1.

At each level of abstraction (after each decomposition) the relationships and interactions between goals can be used to guide the choice of schemas. In a partial order setting, interactions between goals would lead the planner to constrain available orderings.

The dashed box in Fig. 4 represents the items the planner can reason about when decomposing the goal marked X. In this standard HTN model, the planner can reason about all the goals (and hence their interactions) at their current level of abstraction and about the world in its *initial* state. The effects of actions subsequent to the initial state and how goals might interact must be modelled explicitly by the plan knowledge.

2.3.2. Modified HTN goal decomposition

In the adversarial planning architecture presented here, goals are expanded *in time order* using a linearisation. The linearisation is used to make goals concrete as soon as possible and model the effect of the resulting actions in a "world model". This linearisation behaviour contributes to the planner's ability to reason simultaneously and consistently about both abstract (strategic) and concrete (tactical) aspects of the game. As noted above, a world model is not a standard feature of HTN planning systems and goal expansion is quite different from the standard version described above. Goal expansion in the modified approach is shown in Algorithm 2.

```
procedure Refine-Plan-(Modified)
  while Not Empty(Agenda) do
    1. choose a goal NextGoal from Agenda to decompose
    while NextGoal has not resulted in a primitive action do
      2. select a decomposition schema S for NextGoal (taking into
      account the world model)
      3. remove goal NextGoal from Agenda
      if S describes a primitive action then
        4. perform the action in the WorldModel
      else
        choose a goal DescendentGoal from schema S
        5. let NextGoal = DescendentGoal
        6. insert schema (S – DecendentGoal) into Agenda
```

Algorithm 2. This expansion algorithm is similar to the standard expansion algorithm (1) but additionally ensures that one of the subgoals descended from *NextGoal* produces an action in the world before another *NextGoal* is chosen.

There are three key differences between the two decomposition strategies:

- In the modified algorithm, steps 1, 4 and 5 contribute to the linearisation of the plan by forcing the planner to choose a time order for goal expansion (and execution). Once a goal has been chosen for expansion the planner works on its descendent subgoals until at least one of these can be used to choose an action in the world.
- When actions are chosen, their effect is played out in a world model. The information in the world model can subsequently be applied to the choice of plan schema (step 2).
- The planner is performing backwards chaining in the *space of plans* (starting with the top level goals and decomposing) but with an element of forward chaining in the *space of moves.* That is, the initial state and subsequent modelled states clearly show which options are available for the achievement of low level goals. Plans and world state come together to choose the move.

This approach necessarily leads to a total-order planner. At each stage there can be goals of all abstraction levels in the plan, but there must also be a linearisation, and goals are decomposed in time order. Once a primitive action has been performed,

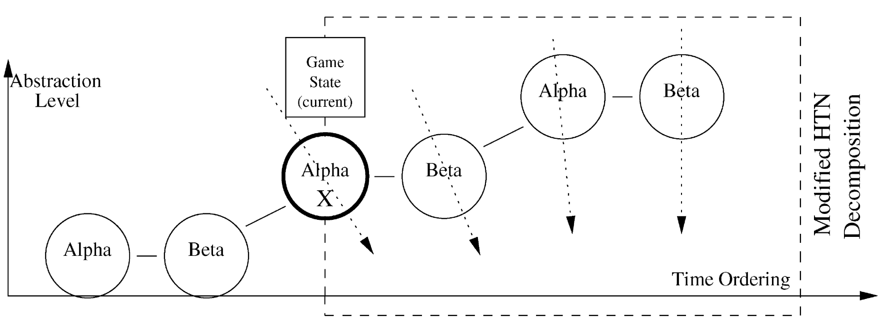

Fig. 5. Goal decomposition in modified HTN planning. The dashed arrows represent the decomposition of goals (eventually) into concrete actions.

the subgoals (generally at various levels of abstraction) which remain on the agenda are retained and may be expanded next time the agent gains control of the planning apparatus.

Fig. 5 illustrates goal decomposition for the modified HTN model. Again, the dashed box represents the items the planner can reason about when decomposing the goal marked X. The decomposition progresses left to right in time order; here the planner can reason about the outstanding abstract goals and about the *current* state of the world.[16] The first two primitive actions (one chosen by Alpha and one by Beta) are now represented in the world model and their effect on the state of play clearly seen.

The planning process can be seen as guiding search using goals, similar to the way Paradise applied chess knowledge (Section 5.4.1). This process helps to choose the moves, but the complex interactions between goals are partly modelled in the world model. As the planner traverses the space of plans (explores the different options for attack/defence that might apply), it also traverses the space of moves. For comparisons with other planning systems see Section 5.4.

3. An adversarial Go planner

The planning architecture was instantiated as a Go reasoning system called GOBI both to test the architecture in a challenging domain and to investigate the usefulness of the goal-driven approach for Go. GOBI consists of a set of knowledge modules which plug into the planning architecture. The knowledge modules provide the Go domain knowledge, plan schemas and goal types which the reasoner can use to solve problems.

3.1. An example Go plan

Fig. 6 shows a problem from Volume I of "Graded Go Problems for Beginners" [49]. The aim is for black to move first and kill the white group of stones. The

[16] Note that the dashed box can be enlarged again upon backtracking.

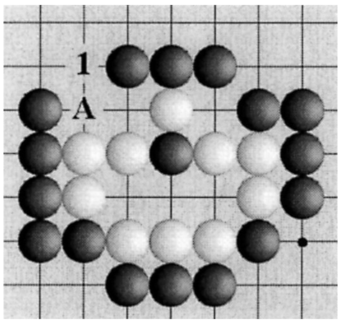

Fig. 6. Example A: black to play and kill the white group.

task is specified to GOBI as two abstract goals: the goal *kill-group* for agent Alpha (playing black) and the goal *save-group* for agent Beta (playing white).[17] Agent Alpha takes control first, decomposing the *kill-group* goal using one of the available plan schemas. An abstract plan for killing this group might be a conjunction of the following subgoals:[18]

- *surround-group* – stop the group from running and connecting.
- *squeeze-space* – reduce the space the group has to live.
- *prevent-eye-formation* – block any attempt by the group to make eyes.[19]

One of these subgoals is then expanded further to the next level and so on until at the lowest level in the hierarchy a move such as *play at A* is chosen to satisfy a simple goal such as *prevent-escape-at-1* (Fig. 6).

Alpha plays the move onto the board in the world model which gives the new world state for Beta to work with. Alpha still has a set of goals at various levels of abstraction remaining in its agenda. These remaining goals represent the plan on how to follow the first move, i.e. which other subgoals/actions need to be achieved to make the plan complete. To validate that this first move is good (in this case playing at A would not be), Alpha must eventually show that all these goals can be achieved no matter what Beta does. These remaining goals are kept by Alpha until after Beta's turn.

Beta now begins by expanding its single goal *save-group* in the context of the new board position (after Alpha playing at A in Fig. 6). A possible plan schema for this goal is:

- *make-eye-space*,
- *make-eyes* (try to form two eyes).

After Beta's move is played into the world model, control is returned to Alpha which then tries to satisfy the rest of its goals. The interleaving of goal expansions by the

[17] The goals need not be directly opposing.

[18] This abstract plan is quite intuitive. It is not obvious how a data-driven system would represent the equivalent of such a plan.

[19] An eye in Go is an enclosed space where the opponent may not play – a group with two eyes is unconditionally alive.

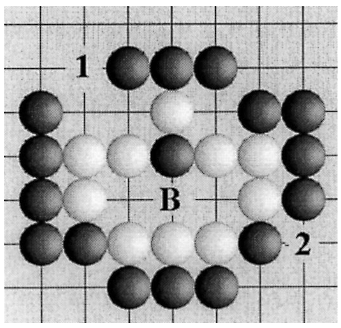

Fig. 7. Example A: GOBI plays at B and this kills the group.

two agents continues until one is able to satisfy all of its aims (and thus implicitly its main aim). The opposing agent is informed of this fact and then backtracks to explore any alternative options it has which might produce a better outcome for itself. In this way a contingency tree is generated which either proves or refutes the validity of the first move (Alpha's).

For this example (Fig. 6), GOBI returns the move at B in Fig. 7, which kills the group. Among the defences tried by Beta are trying to run out at 1 and counter-attacking by playing at 2 (which puts the single black stone in the bottom right hand corner under threat). Since all the moves tried by both agents must be *part of plan of action*, the number of possible moves searched is very small compared to the number of available moves (which is quite large even in this small problem).

3.2. Representing Go knowledge

The planning architecture and Go knowledge modules which make up GOBI are all written in Common Lisp. Around 1400 lines of code make up the plan knowledge (i.e., schemas and goal types) GOBI has. Writing a full-board Go-playing program is a significant exercise in knowledge engineering, so to enable us to add enough depth in knowledge to do useful testing in a short time, GOBI's knowledge is focused on the area of killing and saving groups.[20] The knowledge base is made up of 45 goals at five different levels of abstraction. The average number of applicable plan schemas per goal is approximately two (thus the knowledge has a relatively low branching factor). The two highest level goals available in GOBI are *kill-group* and *save-group*, which emphasises the focus on life and death problems.

The following example plan schemas taken from GOBI's knowledge base illustrate the structure of the knowledge the program holds (the *save-group* goal was also mentioned

[20] This does not mean GOBI is limited to enclosed problems (see Section 4.5).

in the previous example). Note that the plan knowledge is not complete (for example making eyes is not the only way of following up a counter attack); more work is needed to extend the knowledge base.

```
GOAL: save-group,
LEVEL = 5,

 Schema 1 - Find Eyes:
  *make-eye-space,
  *make-eyes.

 Schema 2 - Escape Group:
  *running-sequence,
  *secure-escape.

 Schema 3 - Counter Attack String:
  *locate-vulnerable-string,
  *kill-string,
  *make-eyes.
```

In turn, the *make-eye-space* goal from `Schema 1` has two alternative schemas:

```
GOAL: make-eye-space,
LEVEL = 4,

 Schema 1 - Ambitious Extend:
  *large-extending move,
  *consolidate-space.

 Schema 2 - Creep extension:
  *creeping-extending-move, // A single
  *consolidate-space.       // step extension.
```

The structure of the knowledge shown here is intuitive for Go and is very different from the kind of heuristic information used in data-driven approaches. The knowledge represented can be seen as an AND-OR tree with the AND component represented in the conjuction within the plan schema and the OR component represented in the choice between alternative schemas. Goals within plans are not evaluated at every step. Simple goals are established by finding a satisfying move or using a test. For higher level goals, truth is inferred from the fact that all the goals descended from them were achieved.

3.3. The planning process

This section provides some more details of how the Go instantiation of the planning architecture works.

3.3.1. Choosing top-level goals

To be able to plan the planner needs to have goals to achieve. In GOBI these goals are given by hand. In order to extend GOBI to whole-board play, one of the most important tasks is to define goals and abstract plans above the current highest level. The planner may keep these goals open (and still unexpanded) from one turn to another and indeed for long periods of the game. Since shorter term tactical goals for choosing individual moves are also expressed in terms of goals (these are the goals GOBI already has) this should provide for a smooth transition between the tactical and strategic levels of Go.

The current top levels goals in GOBI are *save-group* and *kill-group* which is quite limiting. Potential additional higher level goals include, for example: *make-moyo*, *prevent-invasion*, *extend-influence*, *extend-territory*, *hold-corner*, right up to *win-game*.

3.3.2. Planning steps

As described in Section 2 during the reasoning process a planning step is made up of two parts:
(1) choosing a current goal to replace in the plan and
(2) choosing an appropriate plan schema to apply to the goal.
In GOBI, the choice of which goal to expand next is unordered, but after choosing a goal from the agenda, GOBI uses a linearisation and continues working on the same goal down through several levels of expansion. Goal expansion follows the algorithm presented in Section 2.3.2. Once a goal has been chosen in step 2 of the algorithm, one of its associated set of plan schemas must be selected for use. The plan schemas in GOBI are tried in a fixed order designed to try the most promising goal decompositions first. Plan preconditions can also be used to screen out schemas not thought to be suitable for the current situation. If one schema leads to a dead-end, backtracking allows GOBI to try the others available for the situation.

The expansion of goals into plan schemas eventually leads to goals at the lowest level of abstraction. These goals need to be checked for satisfaction and used to choose moves. Some example lowest level goals are:
- *fill-a-liberty*,
- *play-a-hand-move* (near *here*),
- *play-a-connecting-move* (between *string1* and *string2*),
- *play-a-placement-move*,
- *play-a-blocking-move* (near *here*),
- *play-an-extending-move* (near *here*).

The plans stored in the Go modules are not preprogrammed solutions and are expanded *in the context of the current game state*. The development of a plan is influenced by schema preconditions and by the choices in the world for making the lower level goals true. The failure of goals early in the plan forces choice of alternative sub plans for making these goals true.

3.3.3. Using a world model to take opportunities

Although the work here focuses on goal-driven approaches to Go, it is clear that human players mix both types of reasoning. Patterns are thought to play a large part in human Go play. Thus, there is a need to consider how the two approaches can be mixed successfully.

One of the advantages of using a world model in the planner is that the changing situation of the state during planning is reflected in the world model. The changing world state may highlight interesting opportunities (or obvious problems) which arise as a result of the plan actions but were not expected effects. The architecture described in Section 2 was extended to include plan *critics* which have access to the world model and watch for important situations. The critics are able to insert goals into the agendas of the agents for the planner to consider in parallel with the current track of reasoning. Thus, the planner is made aware of opportunities or obvious problems during planning and can react to them. [21]

Two critics were added to GOBI which detect groups under immediate threat of capture and insert (optional [22]) group-saving/group-attacking goals into the agents' agendas. The planner may have a good plan of attack but have plan knowledge falling short of suggesting the final killing move and hence the system would normally decide that the plan had failed and try another approach. [23] With the critic switched on, the strong positions from the good attack can be checked to see if there are opportunities for killing a group and the planner told about them. The application of critics increases the size of the search space, but makes the system more robust in situations where the a priori plan knowledge is insufficient.

4. Testing and evaluating GOBI

Yoshinori's four-volume series [49] provides an excellent source of problems for testing Go programs. The books provide problems designed to teach human players how to play Go. Each Go problem gives a board configuration and asks the player to choose the move or moves to achieve a particular aim (such as defending a group). The configurations of stones are usually situations which could arise during normal play but taken out of a full game context. Most problems have a unique solution which is useful for evaluation.

Since setting up each test problem was time-consuming, we chose (test set I) a representative sample (85 problems, approximately one third) of the problems from volume I of [49]. Problems were chosen for the test set to concentrate on the more

[21] As noted in Section 2.3.2 the use of a world model also provides for forward chaining in the space of moves. The forward chaining however is only to model which moves are available for goal satisfaction. The critics here represents situations where the game state prompts *explicit* motivation for a move or new goal.

[22] Optional in this context means precisely that satisfaction of the top level goals is not contingent on also achieving the additional optional goal.

[23] All moves have to be part of a subgoal to be suggested.

complex life and death problems, excluding the sections on simple capturing problems, Atari, Ko, opening, endgame and Seki. Some problems from the Ko and Seki sections were included out of interest.

A second set of tests using problems from volume II [49] and from "Life and Death" [11] was also conducted. All the tests were limited to 30 s of runtime on a SUN Ultraspace.

4.1. Test results

The system successfully solved 74%[24] of the examples in test set I, which is a significant achievement given its limited knowledge and the complexity of some of the problems. GOBI has also solved considerably harder problems from volume II [49, 11]. These further tests were not comprehensive however, and used a range of hand picked examples, so the most useful indicator of performance remains the performance on problems from volume I [49].

In 98% of the correct solutions, GOBI considered the most significant defences or opponent responses in its plan (and refuted them). This statistic is encouraging since not only were the answers correct – so was the reasoning behind them. Most of the failures were due to incomplete plan knowledge. Several problems relied on groups connecting and escaping to live, for example, for which GOBI currently has no plans. Another weak area was in problems which required a lot of forcing moves (e.g. ladders). GOBI has no special way of handling these and so is not able to take the advantage of the problem simplification they provide (they are planned for in the normal way).

Strengthening GOBI's defensive knowledge led to an improvement in attacking plans and vice versa, reflecting the fact that the better opponent model is more likely to find refutations for poor attacking plans. For some of the more difficult problems in [49, 11] reasoning was less complete (fewer of the possible counter strategies were explored). On these harder problems, as GOBI's plan knowledge ran out there was a trend of increasing reliance on the critics (data-driven aspect) for finding solutions. This trend is encouraging since it shows that data-driven and goal-driven approaches can be successfully combined in this way (but less encouraging for our knowledge engineering!).

4.2. The cost of move choice

The cost of move choice can be divided into three components:

(1) *Abstract planning cycles* are very cheap since they consist purely of two cycles of matching items in a list (choose a goal, choose one of its plans). The process can be more expensive if complex preconditions are used to select between plans.

[24] 67% when critics were disabled.

(2) *Checking the satisfaction of low level goals* is also inexpensive, since the goals are very focused (*are these two strings connected*? *could black run out here*?).[25] Checking is made easier by the fact that it can be supported by various representations of the game state – strings, groups, influence etc.

(3) *Using low-level goals to generate moves* is the most expensive part of the process, although the cost is kept down by the fact that the goals by this stage are focused and limited to a small area (the *here* in the section above). In GOBI, selection is done using simple rules which define the type of move which could satisfy a goal.[26] An example set of rules is that a *connecting-move* must:
   (a) be a liberty of (next to) *string1* and
   (b) be a liberty of (next to) *string2*.

Although the workload is divided between these three parts, it is clear that this move choice method is considerably more expensive for choosing a single move to try in the move tree than most data-driven methods. The potential gain of using this method is in *reducing the number of candidate moves in the move tree which need to be tried*, and *simplifying the application of evaluation functions* by restricting both the parts of the board on which they operate, and allowing them to consider only how well a position satisfies a given goal instead of trying to somehow produce an estimate of the absolute value of this position.

For the examples in test set I (see Section 4) there were on average 5.9 planning cycles per move chosen (including the cycles which choose moves). This average was relatively constant over the whole range of problems. To analyse the cost of using planning knowledge in GOBI we converted the planner to run without its plan knowledge and perform depth first search with α–β cutoffs. The α–β search running in the planning framework used the same board data structures, checks for move legality, backtracking mechanism and evaluation functions as GOBI. That is the search used available goals to test for termination of search and for leaf nodes. The analysis showed that given GOBI's current knowledge, each move tried in the game tree involves an average overhead of 2.5μ, where μ is the average cost of a move taken by α–β in this setup. Consequently, the cost of choosing a move in GOBI is about 3.5 times as much as a move made by the search algorithm. We note that although keeping the other features apart from the knowledge constant makes for good comparison (since modules such as the board manager could be implemented much more efficiently) general comparison with α–β search is dependent upon the efficiency of using GOBI's goals as an evaluation function.

Regarding the cost of move choice there are two important tradeoffs in the planning process:

[25] Again, some checks can be more costly such as checking for simple eyes (GOBI currently has a very simple eye recognition module).

[26] Move selection could just as well have been done with local pattern matching. The planning framework poses no restriction on how this relationship between abstract goals and concrete actions is established (in general this is domain dependent).

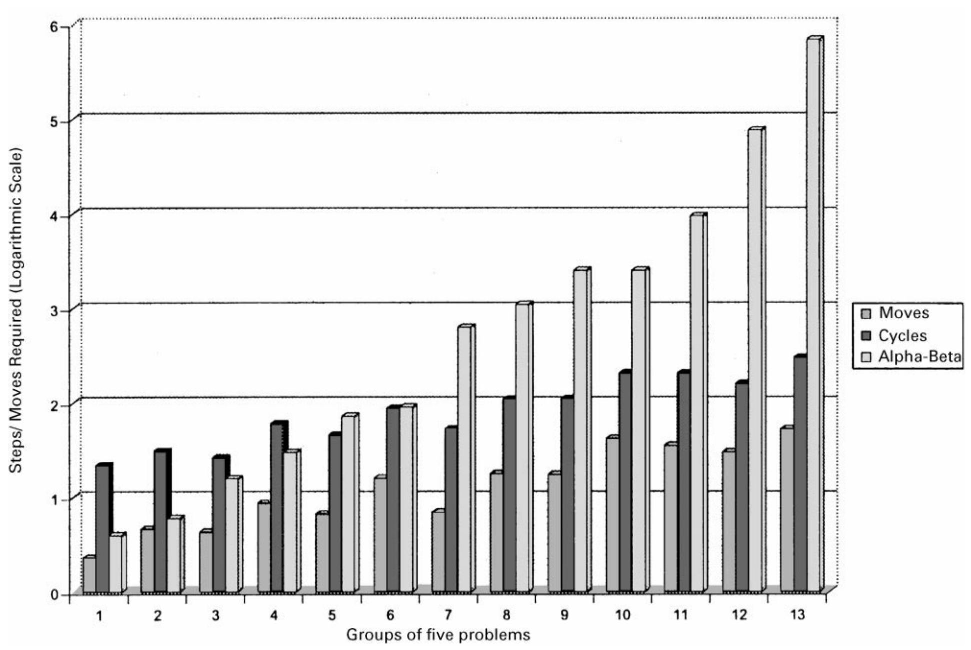

Fig. 8. The number of moves and planning cycles taken by GOBI, plus the estimated α–β search space size for the 63 correctly answered problems from test set I.

- If goal selection and expansion mechanisms get more complex (i.e., through the extensive use of complex preconditions) their time cost will increase, but their applicability will be restricted, reducing search during planning.
- The simpler and more specific the lowest level goals are, the easier it is to establish their truth and choose actions with them but the more planning is needed to decide upon them.

4.3. Number of nodes visited

Many data-driven game playing systems are based on some form of α–β search which makes it a good scheme for comparison with GOBI. The aim of this section is to discuss the number of nodes in the search tree GOBI visits compared to the number a standard α–β search might visit. Unfortunately, the search spaces for the test problems quickly become very large so empirical testing of an α–β algorithm is inaccurate and time consuming. We therefore rely on a theoretical estimate which is derived in Appendix A.

Fig. 8 compares estimates of the size of the search spaces for each of the 63 examples from test set I which GOBI answered correctly (see Section 4.1). The results have been grouped together into sets of five to make the graph more readable.

For each group the graph shows: the average number of moves taken by GOBI (moves), the average number of planning cycles required (cycles) and the size of the estimated α–β search space (alpha-beta). To group the results, the 63 cases were ordered in ascending order of estimated α–β search space (primary criterion) followed by ascending order of number of moves taken by GOBI (secondary criterion). The values were then grouped together in fives according to this order (the first five, the next five, etc.). The full table of these results is available in Appendix B.

Since the graph's vertical axis is on a $\log_{10}$ scale, the graph clearly shows the cost of α–β increases rapidly as the problems become more open (larger search spaces). GOBI, on the other hand, is able to dramatically reduce this rate of increasing cost both in the number of nodes visited and in the number of knowledge cycles. For small search spaces, GOBI requires more planning cycles than α–β visits nodes in the search tree. This result is to be expected since even for simple problems GOBI needs to go through its reasoning process. Since the planning cycles are much cheaper than making moves in the world this extra cost is not a significant problem. More important is the gain in using the plan knowledge as the problems become larger. As a commentary to the results in the graph:

- Already for problems in Group 5 GOBI requires less planning cycles and less moves in the move tree. The problems in this group have search spaces in the order of 90 nodes[27] and GOBI solves them in 3–10 moves.
- Typical results from group 7 include trying 10 moves, with 46 planning cycles for estimated search space 630.
- Typical for group 10 is 28 moves tried with 125 planning cycles (the estimated α–β search has 2520 moves).
- By groups 12 and 13, the difference is already several orders of magnitude. GOBI typically takes around 33 moves to solve problems with 10 or 11 points to play on (and hence estimated α–β search spaces of size greater than a hundred thousand and a million nodes).

The average number of moves tried by GOBI averaged over all 85 problems in the test set is 31 moves (for an average of 174 planning cycles per problem). Averaging over only the 63 correctly answered problems (those shown in Fig. 8) this becomes 24 (with average 144 planning cycles per problem). Giving the α–β average is not very useful since it is dominated by the largest terms.

For the problems that GOBI failed on, 9 used on average number of moves before giving up, 8 resulted in no moves at all (or very few) and the remaining 5 times out after 30 s without having found a solution and having tried between 72 and 410 moves. (A full listing of results is given in Appendix B.)

4.4. Examples from the test sets

To provide a better idea of the kind of problem GOBI was tested on this section gives six commented examples from the test sets. In each case the $*$ symbols in the figure mark the correct first move (black plays first in each case).

[27] See Appendix A for explanation of these estimates.

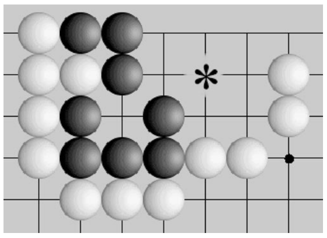

Fig. 9. Example B, black to play and save the group (number 30, test set 1).

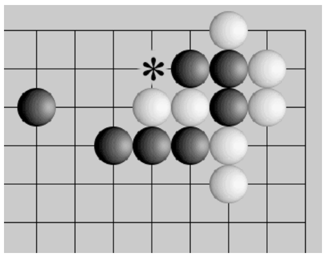

Fig. 10. Example C, black to play and kill the partly enclosed white string (hand-pickled example).

The problem in Fig. 9 was solved in 22 moves and 107 cycles (estimated α–β search space $\approx$ 22 680). The move at $*$ ensures that there is enough room for black to subsequently make a second eye. A determined attack from white can still make saving the group difficult and can go up to 7 moves deep.

Example C (Fig. 10) is harder than it appears. A standard attack on the white string would be to play one point to the left of the $*$ (this attack is called a *net move*). For this problem when GOBI tries the net move, the opponent counter-attacks and kills the black string closest to the edge of the board. The successful counterattack forces GOBI to consider another plan. By playing at $*$, GOBI forces the white string to the left, trapping it against the black stone on the far left. GOBI's move also strengthens the vulnerable black string at the same time (though this is not the primary objective). The plan takes 67 moves and 244 cycles to construct (estimated α–β search space $\approx$ 113 400).

GOBI successfully solves the problem in Fig. 11 using 21 moves and 146 planning cycles (estimated α–β search space $\approx$ 630). The move at $*$ is correct in this case since it is the only way to ensure that black can make two eyes for its group.

Fig. 12 shows a harder problem with several interacting groupings of stones. GOBI takes 60 moves and 294 planning cycles to solve this problem (estimated α–β search space $\sim$ 1 247 400 since there may well be captures during search). The move at * breaks the linkage between two parts and must be followed up by at least one more telling move to ensure that the white group eventually dies.

GOBI fails for both examples F and G. In example F (Fig. 13) it takes 15 moves and 68 planning cycles before giving up (the size of the α–β search space is hard to

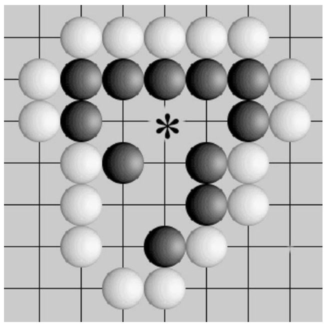

Fig. 11. Example D, black to play and save the group (number 26, test set I).

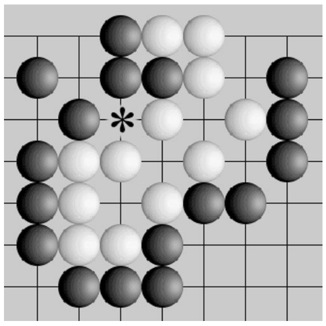

Fig. 12. Example E, black to play and kill the white group (number 48, test set I).

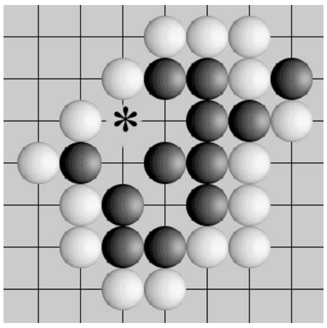

Fig. 13. Example F, black to play and save the group (number 73, test set I).

estimate since the problems involves a possible breakout through the diagonal line of white stones on the top left). This problem is difficult since it relies on a Ko threat being applied.

Failure for example G (Fig. 14) is due to lack of knowledge. GOBI sees all the white stones as a single group and tries a plan to encircle them but gives up without trying any moves. GOBI has no plans for breaking up groups before attacking them. This problem also raises the point that the top-level goal here needs to be redefined

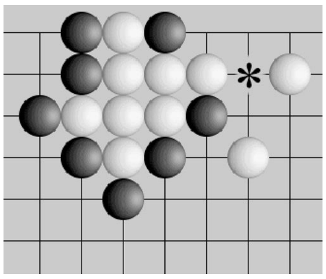

Fig. 14. Example G, black to play and kill (from vol. II [49]).

to allow GOBI to settle for taking just the larger part of the group. (For completeness GOBI takes 0 moves and 5 planning cycles, again search space estimation is difficult.)

To give an idea of how representative these problems are compared to the rest of the test sets: Problems B and D are about average, problems C and E and F are quite hard (C in particular), problem G is easy (although GOBI still fails on it).

4.5. Open examples

Although currently limited by its knowledge base, GOBI is more general than specialist life-and-death programs. Knowledge for other parts of the game can be freely added and GOBI is also able to handle more open positions, such as example H in Fig. 15 (which for example GOTOOLS would have trouble with [48]). One would expect GOBI to compare well against existing systems in open (though still tactical) positions where the number of possible move sequences increases rapidly. The plan knowledge in the system acts to focus on only the relevant parts of the move tree. This effect can be clearly seen in Fig. 8 (Section 4.3) where the cost in moves increases much more slowly than for α–β.

5. Evaluation of the goal-driven approach to Go

As with any method for applying knowledge to searching large spaces, the success of the goal-driven approach depends upon features of the domain of application. This section examines the advantages and disadvantages of applying this approach to Go play.

5.1. Advantages of the goal-driven approach

The goal-driven approach which is presented here has some clear advantages for Go and other similar games. Together with some of the previous work on the usefulness of planning at the strategic level of Go, GOBI shows that this approach can be used for reasoning at all levels. The hope is that it will eventually be possible to move transparently between tactical and strategic levels of play and providing a unifying framework for Go.

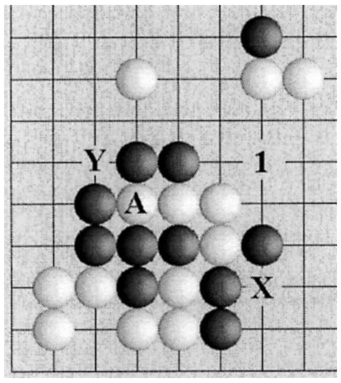

Fig. 15. Example H: GOBI solves this more open position, killing the white string marked A by playing a net move at 1. Adding white stones at X or Y for example (or various other places) would cause GOBI to realise that the string can no longer be killed directly.

5.1.1. Representation and communication of domain knowledge

Go knowledge in books and folklore is often expressed in a form appropriate for encoding as decompositions of abstract goals into other goals at different levels of abstraction. As reported in [35], there is a rich vocabulary which Go players can use to express their reasons for making a move. There are many popular proverbs, which convey strategies at various levels of abstraction, for example "death lies in the hane" is more tactical, whereas "don't push along the fifth line" is quite strategic. It may be easier to add this kind of knowledge to a goal-driven system than to a data-driven system which requires the delicate balancing of heuristics.

Adding knowledge in the form of plans is made very much easier by the existence of natural levels of abstraction when describing the game state. Game positions are often considered not only in terms of stones but in terms of macro structures such as strings groups and areas of influence.

By following the trace of the goal decompositions one can see *why* the Go player is trying to do something – its aims and plans. Access to aims and plans is not only helpful for adding knowledge and debugging, but could be useful in the context of a Go tutoring aid. Some of the current commercial Go systems (MANY FACES OF GO for example) have teaching mechanisms but it is not clear whether these mechanisms are based directly on the reasoning process of the computer player, or are instead based on a high-level post-hoc analysis (which therefore will not accurately reflect the reasoning which led to this choice of move).

5.1.2. Search properties

Using a goal-driven approach leads to a very different search from a data-driven approach. Some key advantages are:

- There is less need for global evaluation functions: Full board evaluation reduces to checking if lower level goals can be satisfied. Evaluation functions may still be used in a limited way to carry out judgements which cannot be made on a move-by-move level, for example using an influence function to judge whether or not a group can successfully run, or using a fast α–β search to determine the life or death of enclosed groups. When an evaluation function is employed in determining whether or not a goal has been satisfied, the goal can be used to provide a focus for the evaluation function to a limited area of the board. The use of goals to focus evaluation makes sense when thinking about how humans play go – often making instinctive judgements (evaluations) of a local situation but rarely considering (evaluating) the whole board.

- Quiescence (a problem for many game-playing programs) is defined automatically, thus avoiding problems such as the horizon effect in search. Since each agent has an agenda of goals, a situation can be defined as "open" (unsettled) until all goals are achieved.
- What seem like heuristically bad moves (e.g. sacrifices) are not discriminated against because the only consideration is their value to the plan.

While these advantages are important, perhaps the overriding advantage often associated with goal-driven approaches is their resilience in domains with large search spaces and branching factors. As the figures in Section 4.3 show, the planner dramatically reduces the number of nodes searched in the move tree. The advantage is still significant even if the extra cost of planning is taken into account.

5.2. *Disadvantages of goal-driven search*

Obviously the goal-driven approach is not always the best choice and has its own difficulties:

- The goal-driven approach requires significant effort to encode strategies as goal decompositions. By contrast, in the data-driven approach, good play can be achieved using even relatively simple evaluation functions if the game tree can be searched deeply enough.
- There are some types of knowledge which are hard to express in a goal/plan oriented framework, such as knowledge which is not reliant on understanding the motivation behind a move (patterns for example). It seems clear that good Go play requires both pattern (data-driven) and abstract plan (goal-driven) knowledge which is what leads us to try and integrate the two approaches (see Section 3.3.3).
- For games with low branching factors, shallow search trees or where near exhaustive search is possible data-driven approaches have a strong advantage. It is only when searching most of the move tree is infeasible and large amounts of knowledge are needed to prune the tree that goal-driven approaches become more useful.[28]

[28] This point was illustrated by PARADISE [44] in the early eighties which despite being able to solve Chess problems requiring search up to 20 ply deep (far beyond other Chess programs of the time), still saw its knowledge-based approach outstripped by the ever increasing efficiency of fast search techniques.

The main problem with GOBI itself is still lack of knowledge: much more is needed before GOBI could be used as a tool to play the complete game of Go. Unfortunately adding knowledge takes a long time since it must be hand coded; most of the top programs have had knowledge added over periods of years. In this respect the goal-driven approach (and many current approaches) are at a disadvantage to learning or non-symbolic approaches which can use automatic training to improve.

5.3. Scaling issues

As with all knowledge based approaches, the search space is determined by the knowledge in the system. The key to making knowledge based approaches work is good organisation of that knowledge. The size of the search space is determined by the number of goal schemas, and by how much they overlap. A small number of goal schemas, which are mutually exclusive, lead to a relatively small search space. Conversely, a very large number of goal schemas which overlap in many ways lead to a large search space. In practice, the size of the search space lies between these two extremes.

Schemas which do not overlap, for example schemas which address different parts of the game, can be added without adversely affecting performance. Adding more detailed plans needs to be done carefully, since it can lead to overlaps and redundant search. The levels of abstraction used in GOBI (and in many AI planners) are key in avoiding redundancy since knowledge can be appropriately structured and similar plans can be grouped together and expressed in a compact form. Badly structured or overlapping knowledge can cause similar problems to occur in data-driven approaches where pattern matching is used to suggest moves.

In Wilkins' PARADISE [44] work, as in our work, there was a worry that extending the system by adding new knowledge would have adverse affects on system performance. New production rules were added to PARADISE in order to allow it to solve problems outside the initial development set [44, p. 193]. The new problems were correctly solved, and system performance was not degraded. In fact, the extended system now found *better* plans for some of its previously solved positions. This result is heartening for our own work.

5.4. Comparisons with other approaches to adversarial planning

One of the advantages of building a generic planning framework (Section 2.2) and then separately instantiating this to play Go (Section 3) is that the characteristics of the plannar are clearly laid out. This separation improves understanding of the planner's behaviour and facilitates comparison with systems from other domains.

The most useful comparisons can be made using a classification based up on the expansion scheme used during planning, since this is the defining feature of any HTN planning system. The planning systems can be classified as follows:

- *Total order decomposition*: PARADISE, TIGNUM2, GOBI, the battlefield management system in [1].

- *Partial order decomposition*: Work in [21, 23, 37] and the Interim.2, and Gobelin programs.

In this classification and leaving aside the application domain, gobi is most similar to Paradise and Tignum2. In fact these planners not only use a form of total order expansion but also incorporate aspects of forward chaining (Section 2.3.2).

5.4.1. Paradise

The idea of using goals to guide search originally comes from [29] and was then incorporated into Paradise. Although Paradise is based on rule-based expert systems and in this sense is quite far removed from HTN planning, its search approach leads to a type of goal guided search at two levels of abstraction. Each goal is expanded by a rule based knowledge source in the context of its final place in the plan. This guided search model is conceptually very similar to the modified HTN expansion scheme described in Section 2.3.2. There are many differences between Paradise and the planner used in gobi, but perhaps the most significant at a conceptual level are that:

- Paradise only used two levels of abstraction – the ground space (moves) and one abstract level (goals or concepts). With its current knowledge gobi uses five levels of abstraction.
- Paradise's starting plans are formed from a static evaluation of the initial game state and global static evaluations may be applied at various points throughout the search. In gobi, goals can have associated information about what information to apply from the environment for expansion (in the form of preconditions). This precondition mechanism gives a full range of flexibility (from very focused detailed evaluations of a local area for low-level goals to very sketchy high level evaluations of the whole board for high level goals).
- Paradise can analyse failed plans and use the extracted information in constructing a new plan.
- Paradise uses different models for attacking and defending players. While the attacker uses the plan suggested by static analysis to focus on only a few candidate move choices, the defender tries all "reasonable" moves. In gobi, attack and defence use the same plan schemas. Indeed, both players may be attacking simultaneously.

Above all, the planning architecture used by gobi provides a simple generic framework which is simpler to understand than the mixture of different elements which come together to make up Paradise.

5.4.2. Tignum2

Tignum2 [40, 39] also has a decomposition model very similar to that used in gobi (and is explicitly based upon HTN planning). The propositional logic formulas used by Tignum2 to represent the state of play directly correspond to the use of a world model in gobi. The most significant difference between gobi's planner and the one used in Tignum2 is in the expressiveness of the planning language. Tignum2's planner is kept total order by restricting *plan knowledge* to being totally ordered, preventing subgoals

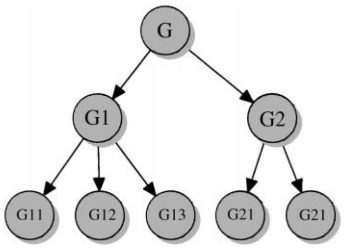

Fig. 16. Goal decomposition tree: the top level goal G decomposes into two subgoals G1 and G2, each of which is again decomposed.

descended from different higher level goals from being interleaved.[29] In TIGNUM2, given the decomposition tree shown in Fig. 16, goals descended from G2 could not be interleaved with goals descended from goal G1. Furthermore the goals descended from G1 must be fulfilled in the order given in the schema – G11, then G12, then G13.

There is no such restriction in the planning framework presented in this paper: subgoals can be interleaved and all linearisations of subgoals are valid.[30] This flexibility preserves the expressiveness of standard HTN planning (which is semi-decidable, see [14]) while TIGNUM2's planner is strictly less expressive (EXPSPACE-hard and in DOUBLE-EXPTIME, [27]). The extra flexibility in GOBI's planner is crucial for games such as Go where interleaving goals is a natural part of play. In Go, capturing a group for example may involve several tasks (surrounding it, preventing eyes etc.) which need to be achieved in pseudo-parallel. That is, concentrating on thoroughly surrounding a group will result in the opponent making eyes and saving the group (and vice versa). Steps need to be taken towards achieving both subgoals while stopping the opponent countering either (having enough spare moves to do so).

5.4.3. *Using total order expansion in adversarial domains*

For many practical problems the total order decomposition systems identified in Section 5.4 above have significant advantages over those using partial order decompositions. This observation runs contrary to conventional wisdom in the AI planning community. The least commitment aspect of partial order decompositions was always thought to bring significant advantages. Recent work in [27] discusses the assumptions underlying the use of partial order, backwards chaining strategies for HTN planners and suggests that they are not always valid (and in particular do not apply for Bridge play and process planning). In the context of an adversarial planning domain we can elaborate on these reasons (and include observations noted in [46]).

As is noted in [14] "handling interactions among non-primitive tasks [goals] is the most difficult part of HTN planning". In adversarial domains these difficulties are compounded. More precisely:

[29] This definition of total order planning is slightly different to the one used in [25] and that adopted in this paper.

[30] Constraints can be placed on orderings, but these constraints form part of the explicit plan knowledge.

- *The interactions are significant*: Goal expansion is usually heavily dependent on the game state. The simplifying assumption often taken in AI planning research that expansion choices (*how* to do something) and ordering choices (*when* to do it) are independent [2] is less valid in adversarial domains.
- *More of the interactions need to be taken into account*: In an adversarial domain, the adversaries have control over some of the actions, therefore not only must there be at least one valid ordering, the opponents must not be able to order their actions in such a way as to make the plan fail. The planner must explore many more of the options and orderings possible to ensure they all lead to a desirable outcome.
- *The interactions are difficult to model*: The interactions between moves in a game are generally *designed* to be complex. So complex, in fact, as to fool and entertain human players. Effects are often cumulative, subtle and only obvious over long sequences of moves. This complexity makes interactions inherently hard to model and even harder to model in the abstract where much of the detail is stripped away. In his 1979 paper [43] Wilkins observes that defining abstract spaces for Chess is very difficult because "small details are very important".

Games tend to be built around the effects of complex interactions between the moves of the players. It becomes increasingly difficult to model all these interactions adequately at higher levels of abstraction. If important details (interactions) are not somehow modelled in the abstract space and taken into account during planning, plans will repeatedly fail at the lowest level and cause repeated backtracking.

The use of a total order scheme which is able to model interactions between moves and goals directly in a world model is important in avoiding some of these problems. It also means that an aspect of forward chaining can be introduced at the move level to reduce the number of options available (since the starting state is always known even though the goal state may not be perfectly described). These differences appear to be critical in enabling total order planners to outperform standard HTN planners in tight tactical situations.

6. Conclusions

In this paper we have presented an adversarial planning architecture capable of reasoning about games, and an application of this architecture to Go. The planning architecture and Go reasoner reported here represent an advance on previous work for goal-driven planning in Go. The system:

- Has a clear separation of domain knowledge from the abstract planning architecture and a clear model of the opponent in the game.
- Can reason at multiple levels of abstraction simultaneously.
- Can address complex tactical situations as well as high level strategic problems.
- Can provide support for the integration of data-driven and goal-driven approaches.

We presented the advantages that a goal-driven approach could have for Go. GOBI as a prototype is certainly no match for current systems which play the full game of Go, but is does represent a step towards understanding how goal-driven approaches can be applied to Go, even at a tactical level.

We believe that GOBI demonstrates that an approach based on adversarial planning can be effective for Go. The two main limitations of GOBI as a Go-playing program are:

- The lack of top-level goals. We have indicated how GOBI could be extended with high-level persistent goal which would enable it reason about the entire game.
- The small knowledge base. The knowledge base needs to be very significantly extended in order to make a realistic Go program.

Go has several strong domain features which make goal-driven approaches applicable: very large search spaces, clear layers of abstractions in domain descriptions (stones, strings, groups, etc.), and a wealth of knowledge similar in structure to abstract plans. GOBI represents a further step towards turning this theoretical possibility into a reality. The work described in this paper again shows that Go is an excellent test bed for Artificial Intelligence research. There has been very little work on adversarial planning in recent years – the challenge of Go really motivated this work.

We have outlined why GOBI outperforms previous Go planners at tactical play. The total order decomposition scheme and use of a world model were key in making progress. Taken together with previous work on Chess and Bridge this suggests that the techniques we have described in this paper should be strongly considered when applying planning techniques in future game playing systems.

Acknowledgements

The first author is very grateful for financial support from the EPSRC under grant number 96417312. The second and third authors are supported by EPSRC grant GR/L/11724 and GR/M/45030. The fourth author is partly funded by DARPA/AFRL contract F30602-95-1-0022. Special thanks also go to Martin Müller for suggestions regarding test data, and to Neil Firth for numerous instructive games of Go. We are grateful to the anonymous referees whose comments have resulted in numerous improvements to previous versions of this paper.

Appendix A. Theoretical estimate of α–β search space

As stated in Section 4.3 estimating the size of α–β search spaces for given problems empirically is difficult since search spaces and search times rapidly increase. The performance of α–β is also heavily dependent on the evaluation function used. This difficulty forces us to rely on theoretical estimates. A number of papers published by UCLA in the late 1970s and early 1980s give theoretical analyses of the α–β algorithm's performance. Many of these results are summarised in [28]. However, the results generally rely on a constant branching factor (which in tight tactical Go problems is not valid) and sampling leaf node values from a continuous distribution. For a theoretical analysis closer to the type of problem GOBI solves we need to go a little outside these results.

The following assumptions allow a simple treatment:

- *Assumption 1*: The number of available moves is reduced by one each round and both players play moves from the same finite set of moves. These assumptions do not always hold for Go since captures can increase the number of moves available, some moves may only be legal for one player and players may choose to pass, however they are acceptable for a simple treatment.
- *Assumption* 2: There is only one solution (i.e., one correct move to play first which leads to a win being possible in all of the subtrees following this move). This assumption also does not hold for general Go play but is valid in the type of problems in the test sets.

Considering a search with n options at the top level, Assumption 1 gives the total number of leaf nodes as $n!$ where leaf nodes are defined as the nodes at depth n (or equally $n-1$ in the search). We now estimate the number of leaf nodes visited, on average, by an α–β search algorithm which is able to evaluate leaf states as *win* or *lose*, does not evaluate intermediate (non-leaf) nodes, and stops once it finds the correct first move (as GOBI does).

Over all orderings of moves and solutions for a given problem one would expect the algorithm on average to choose the correct move having tried half the possible moves (contingent on Assumption 2). Following a move selection, *all* the opponents reponses need to be tried to ensure the move leads to a win everywhere, thus making the number of leaf nodes visited equal to

$$\frac{n}{2} \times (n-1) \times \frac{n-2}{2} \times (n-3) \times \cdots \tag{A.1}$$

$$\text{which gives: } \left(\frac{1}{2}\right)^{n/2} n!. \tag{A.2}$$

(The odd n are those multiplied by the factor half.) It is interesting to note that the pruning power ($2^{n/2}$) of α–β is not simply a linear factor of the search space size (see the table in Fig. 17). For an accurate comparison in the number of *moves* tried by the search, we also need to count the number of moves tried in reaching the leaf nodes. The number of moves required becomes

$$\left(\frac{1}{2}\right)^{n/2} n! + \frac{\left(\frac{1}{2}\right)^{(n-1)/2} n!}{(n-(n-2))!} + \frac{\left(\frac{1}{2}\right)^{(n-2)/2} n!}{(n-(n-3))!} + \cdots \tag{A.3}$$

Note again that this figure is valid in the limit, but generally the power of the pruning factor is rounded down to the nearest integer, reflecting the fact it alternates on the levels with odd n. In the limit the new value in Eq. (A.3) is round double the value given in Eq. (A.2). Thus, the number of nodes visited is dominated over n by the number of leaf nodes visited, $(\frac{1}{2})^{n/2} n!$.

Options	Raw	Purnning Factor	E[Leaf nodes]
3	6	2	3
4	24	4	6
5	120	4	30
6	720	8	90
7	5040	8	630
8	40320	16	630
9	362 880	16	22680
10	3 628 800	32	113400
11	39 916 800	32	1 247 400
12	479 001 600	64	7 484 400
13	6 227 020 8000	64	97 297 200

Fig. 17. Calculating the expected number of leaf nodes searched. The column headed *Options* is the number of moves considered in the initial position, *Raw* is the size of the unpruned search space, *Pruning Factor* is the factor by which α–β search reduces the number of leaf nodes visited, and E[*Leaf nodes*] is the expected number of leaf nodes visited.

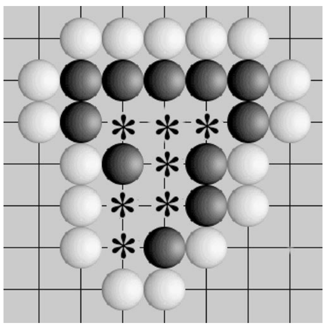

Fig. 18. A mask for a Go problem from test set I. The * symbols indicate the area that needs to be considered for search.

Analysis of the number of moves tried becomes considerably more complex if the algorithm can evaluate intermediate (non-leaf) nodes since the performance of α–β algorithms is generally sensitive to the evaluation function used. We have not taken this possibility into account in our analysis.

The result obtained for the pruning power of α–β on this type of problem (a binary set of state values, one correct move, stopping on finding this move and having a steadily decreasing branching factor), seems to concur with the UCLA results [28] which also predict low order exponential pruning power for their problems.

A.1. Estimating the search space for Go problems

To estimate the search space that would be required by α–β search for the Go problems in the test set each problem has a search area mask defined for it. The mask

Table 1

N	Ref	P	M	Res	E	N	Ref	P	M	Res	E
1	41	42	8	C	6	44	145	702	126	C	2520
2	42	20	3	C	30	45	147	49	7	C	30
3	43	17	2	C	6	46	148	1002	410	F	30
4	44	49	7	C	6	47	149	285	44	C	113 400
5	45	30	4	C	6	48	150	294	60	C	1 247 400
6	99	184	35	C	2520	49	151	20	3	C	30
7	100	74	13	C	90	50	152	22	3	F	30
8	101	75	19	C	90	51	153	47	7	F	90
9	102	70	14	C	30	52	154	9	1	F	90
10	103	176	39	C	113 400	53	155	20	3	C	90
11	104	25	4	C	2520	54	156	61	12	C	2520
12	105	136	22	F	90	55	157	120	20	C	1 247 400
13	106	41	4	C	3	56	158	33	5	C	2520
14	107	38	10	C	6	57	159	62	22	C	2520
15	109	132	18	C	2520	58	160	1002	155	F	2520
16	110a	46	10	C	630	59	161	32	7	F	1 247 400
17	110b	1002	164	F	2520	60	162	1002	151	F	113 400
18	111	724	138	C	2520	61	163	122	18	C	2520
19	146	210	35	F	630	62	164	23	2	F	3
20	121	17	2	C	3	63	165	434	54	C	2520
21	122	42	7	C	30	64	166	609	110	C	2520
22	123	106	12	C	30	65	167	125	28	C	2520
23	124	29	4	C	90	66	168b	21	2	C	6
24	125	116	20	C	2520	67	169b	31	3	C	6
25	126	54	6	C	30	68	170a	20	2	C	3
26	127	146	21	C	630	69	172	17	6	F	3
27	128	93	16	C	90	70	173	11	1	F	6
28	129	279	59	C	630	71	108	4	1	F	3
29	130	77	13	C	90	72	176	23	5	F	1 247 400
30	131	107	22	C	22680	73	177	68	15	F	113 400
31	132	20	2	C	30	74	178	90	19	F	630
32	133	129	20	C	90	75	222	12	2	F	90
33	134	55	8	C	90	76	223	118	23	C	22680
34	135	36	5	C	630	77	224	20	2	F	6
35	136	90	13	C	630	78	225	59	12	C	22680
36	137	18	5	C	6	79	226	767	118	C	630
37	138	54	6	C	630	80	227	518	53	C	22680
38	139	65	18	C	113 400	81	228	103	17	F	90
39	140	93	9	C	630	82	230	56	10	C	90
40	141	50	6	C	630	83	231	12	1	F	30
41	142	113	14	C	30	84	232	35	5	C	6
42	143	87	10	C	22680	85	233	1002	72	F	2520
43	144	814	147	C	1 247 400						

defines which points on the board need to be tried as moves by the search in order to find a satisfactory solution (meaning one which chooses the correct move but also encompasses checking all significant defences). Fig. 18 shows a mask defined for one of the problems from test set 1. The * characters mark the points which define the limited search space.

For our estimates all the masks were defined by hand. Once the mask has been defined for a problem, the number of points defined in it gives the number of moves available in the search space (by Assumption A1 in the previous section). Some masks also included occupied points where stones were captured as part of attack or defence. Together with the formula in Eq. (A.2) (or Fig. 17), this gives the estimate of the expected search space for that problem. We use Eq. (A.2) rather than Eq. (A.3) since the number of leaf nodes is the dominating term.

This estimation of search space size for a Go problem is clearly very conservative since the mask and the search depth are in fact complex to compute automatically. Arguably finding this information forms part of the search problem itself. The advantage of using the hand derived mask and search depth however, is that they give a clearly defined and understandable way of generating estimates. Without these restrictions, search space estimates which could be made almost arbitrarily large and be of little use for comparison.

Appendix B. Test results

For completeness we include a listing of the results for the 85 problems tested in test set I (results are from tests using critics). In Table 1, the column headings are: the problem number (N), the problem number given in [49], the number of planning cycles GOBI takes to plan the problem (P), the number of moves GOBI tried (M), whether GOBI succeeded in finding a correct problem solution (Res, C = correct answer, F = fail), and the estimated number of moves in the α–β search tree for this problem (E). For details of how these examples were selected from [49] see Section 4.1.

References

[1] C. Applegate, C. Elsaesser, D. Sanborn, An architecture for adversarial planning, IEEE Trans. Systems Man Cybernet. 20 (1) (1990) 186–294.

[2] A. Barrett, D.S. Weld, Partial-order planning; evaluating possible efficiency gains, Artificial Intelligence 67 (1) (1994) 71–112.

[3] H.J. Berliner, A chronology of computer chess and its literature, Artificial Intelligence 10 (1978) 201–214.

[4] R. Bozulich, Second Book of Go, The Ishi Press Inc, 1987.

[5] D.J.H. Brown, S. Dowsey, The challenge of go, New Sci. 81 (1979) 303–305.

[6] J. Burmeister, J. Wiles, An introduction to the computer go field and associated internet resources, Tech. Rep. The University of Queensland, January 1997. Available online at: http://www/psy.uq.edu.au/ ~ jay/go/go_page.html.

[7] J.G. Carbonell, Counterplanning: a strategy based model of adversarial planning in real world situations, Artificial Intelligence 16 (1) (1981) 295–329.

[8] T. Cazenave, Système d’Apprentisage par Auto-Observation. Application au Jeu de Go, Ph.D. Thesis, L’Université Paris 6, 1996.

[9] T. Cazenave, Metaprogramming forced moves, in: H. Prade (Ed.), Proc. 13th European Conf. on Artificial Intelligence (ECAI’98), Wiley, New York, 1998, pp. 645–649.

[10] S.F. Da Silva, Go and genetic programming, playing go with filter functions, Master’s Thesis, Leiden University, Netherlands, 1996. Available online at: http://www.wi.leidenuniv.nl/MScThesis/dasilva.html.

[11] J. Davies, Life and Death, The Ishi Press Inc, 1978.
[12] P. Donnelly, P. Corr, D. Crookes, Evolving go playing strategy in neural networks, AISB Workshop in Evolutionary Computing (1994).
[13] M. Enzberger, The integration of a priori knowledge into a go playing neural network, Tech. Rep., University Munich, 1996.
[14] K. Erol, D. Nau, J. Hendler, HTN planning: complexity and expressivity, Proc. AAAI'94, July 1994.
[15] K. Erol, D. Nau, J. Hendler, UMCP: a sound and complete planning procedure for hierarchical task-network planning, Proc. AIPS-94, June 1994.
[16] D. Fotland, Knowledge representation in the many faces of go, Tech. Rep., American Go Association, 1993. Available online at: ftp://bsdserver.ucsf.edu/Go/comp/mfg.Z.
[17] D. Fotland, A. Yoshikawa, The 3rd FOST cup world-open computer-go championship, ICCA J. 20 (4) (1997) 276–278.
[18] I. Frank, Search and planning under incomplete information: a study using bridge card play, Ph.D. Thesis, Department of Artificial Intelligence, University of Edinburgh, 1996.
[19] I. Frank, D. Basin, A. Bundy, An adaptation of proof-planning to declarer play in bridge, Proc. 10th European Conf. on Artificial Intelligence (ECAI'92), Vienna, Austria, 1993, pp. 72–76. Longer Version available from Edinburgh as DAI Research Paper No. 575.
[20] M.L. Ginsberg, GIB: steps toward an expert-level bridge-playing program, Proc. 16th Internat. Joint Conf. on Artificial Intelligence (IJCAI'97), 1999.
[21] S. Hu, Multipurpose adversary planning in the game of go, Ph.D. Thesis George Mason University, 1995.
[22] T. Kojima, K. Ueda, S. Nagano, An evolutionary algorithm extended by ecological analogy and its application to the game of go, Proc. 15th Internat. Joint Conf. on Artificial Intelligence (IJCAI'97), 1997, pp. 684–689.
[23] P. Lehner, Strategic planning in go, in: M.A. Bramer (Ed.), Computer Game Playing: Theory and Practice, Ellis Horwood, Chichester, 1983,, pp. 167–176.
[24] D. Lichtenstein, M. Sipser, Go is polynomial-space hard, J. ACM 27 (2) (1980) 393–401.
[25] S. Minton, J. Bresina, M. Drummond, Total-order and partial-order planning: a comparative analysis, J. Artificial Intelligence Res. 2 (1994) 227–262.
[26] M. Müller, Computer go as a sum of local games: an application of combinatorial game theory, Ph.D. Thesis, Swiss Federal Institute of Technology, Zurich, 1995.
[27] D.S. Nau, S.J.J. Smith, K. Erol, Control strategies in HTN planning: theory versus practice, Proc. Innovative Applications of Artificial Intelligence Conf. (in conjunction with AAAI'98), AAAI Press, 1998, pp. 1127–1133.
[28] J. Pearl, The solution for the branching factor of the α–β pruning algorithm and its optimality, Research Paper UCLA-ENG-CSL-8019, University of California, Los Angeles (UCLA), June 1981.
[29] J. Pitrat, A chess combination program which uses plans, Artificial Intelligence 8 (1) (1977) 275–321.
[30] W. Reitman, B. Wilcox, The structure and performance of the INTRIM.2 go program, Proc. Internat. Joint Conf. on Artificial Intelligence (IJCAI'79), 1979, pp. 711–719.
[31] P. Ricaud, A model of strategy for the game of go using abstraction mechanisms, Proc. 15th Internat. Joint Conf. on Artificial Intelligence (IJCAI'97), 1997, pp. 678–683.
[32] N. Richards, D. Moriarty, R. Miikkulainen, Evolving neural networks to play go, Tech. Rep., The University of Texas at Austin, 1997.
[33] J.M. Robson, The complexity of go, Tech. Rep. TR-CS-82-14, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, October 1982. Also published IFIP: International Federation of Information Processing, 1983.
[34] E.D. Sacerdoti, A Structure for Plans and Behaviour, Elsevier, Amsterdam, 1977.
[35] Y. Saito, A. Yoshikawa, Do go players think in words? — Interim Report of the analysis of go player's protocols, in: H. Matsubara (Ed.), Proc. 2nd Game Programming Workshop, Computer Shogi Association, 1995.
[36] Y. Saito, A. Yoshikawa, An analysis of strong go-players' protocols, in: H. Matsubara (Ed.), Proc. 3rd Game Programming Workshop, Computer Shogi Association, 1996.
[37] P.T. Sander, D.J.M. Davies, A strategic approach to the game of go, in: M.A. Bramer (Ed.), Computer Game Playing: Theory and Practice, Ellis Horwood, Chichester, 1983, pp. 152–166.
[38] J.J. Smith, D.S. Nau, Strategic planning for imperfect information games, Games: Planning and Learning, Papers from the 1993 Fall Symp., AAAI Press, 1993, pp. 84–91.

[39] S.J.J. Smith, D.S. Nau, T.A. Throop, A planning approach to declarer play in contract bridge, Comput. Intelligence 12 (1) (1996).

[40] S.J.J. Smith, D.S. Nau, T.A. Throop, Total-order multi-agent task-network planning for contract bridge, Proc. AAAI'96, 1996, pp. 108–113.

[41] J.A. Storer, On the complexity of chess, J. Comput. System Sci. 27 (1) (1983) 77–100.

[42] A. Tate, Generating project networks, Proc. 5th Internat. Joint Conf. on Artificial Intelligence (IJCAI'77), 1977.

[43] D.E. Wilkins, Using plans in chess, Proc. 6th Internat. Joint Conf. on Artificial Intelligence (IJCAI'79), Tokyo, Japan, 1979, pp. 960–967.

[44] D.E. Wilkins, Using patterns and plans in chess, Artificial Intelligence 14 (1) (1980) 165–203.

[45] S.N. Willmott, Adversarial planning and the game of go, Master's Thesis, Department of Artificial Intelligence, University of Edinburgh, September 1997.

[46] S.N. Willmott, A. Bundy, J.M. Levine, J.D.C. Richardson, Adversarial planning in complex domains, Tech. Rep., Department of Artificial Intelligence, University of Edinburgh, January 1998. Research Paper Number 889.

[47] T. Wolf, The program go tools and its computer-generated Tsumego database, Proc. 1st Game Programming Workshop, Computer Shogi Association, 1994, pp. 84–96.

[48] T. Wolf, About problems in generalizing a Tsumego program to open positions, in: H. Matsubara (Ed.), Proc. 3rd Game Programming Workshop, Computer Shogi Association, 1996.

[49] K. Yoshinori, Graded Go Problems for Beginners, Vols. I–IV, The Ishi Press Inc, 1985.

[50] P.R. Young, P. Lehner, Applications of a theory of automated adversarial planning to command and control, IEEE Trans. Systems Man Cybernet. 16 (6) (1990) 186–294.