Knowledge Based Planning in Games

Prasad Tadepalli

May 30, 1989

CMU-CS-89-135

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper explores the issue of planning in two-person games using approximately learned knowledge which is in the form of macros. A planning technique called Knowledge Enabled Planning, which consists of *generating* new plans by instantiating and composing the macros and *testing* them against the opponent's counter-plans, is introduced using a program called LEBL that learns and plans in two-person games. Some empirical results of testing LEBL in king and pawn endings in chess are presented along with a complexity analysis of the planning algorithm.

This research was partially supported by the National Science Foundation under Contract Number IRI-8740522, and the Defense Advanced Research Projects Agency under Contract Number N00014-85-K-0116.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation, the Defense Advanced Research Projects Agency, or the U.S. Government.

Knowledge Based Planning in Games

Prasad Tadepalli
(Prasad.Tadepalli@cs.cmu.edu)
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, and
Department of Computer Science, Rutgers University, New Brunswick, NJ 08903

1 Introduction

The problem addressed in this paper originally arose in the context of learning. In many domains like two-person games, it is computationally intractable, if not impossible, to learn complete and correct knowledge. Learning systems typically solve this problem by making various *simplifications* and *approximations* to the learning process [7, 1, 5, 14]. However, this introduces errors and imperfections in the learned knowledge and raises difficulties to the planner which makes use of that knowledge. Hence, in systems that incrementally acquire approximate knowledge, either by learning or through human interaction, it is imperative that the planner is able to make effective use of the knowledge even when the knowledge is not complete and correct.

In this paper, I address the problem of knowledge based planning in two-person games, where, due to the inherent uncertainty in the future actions of the opponent, it is intractable to learn complete and correct knowledge. There are several requirements a knowledge based planner must fulfill to be useful in game domains. In particular,

1. It should be able to cope with imperfect knowledge. A weak planner that simply instantiates the learned knowledge is inadequate in complex domains where the knowledge is only approximately correct.
2. However, it is unreasonable to expect the planner to be completely insensitive to the imperfections in its knowledge. While it is allowed to make some errors due to its imperfect knowledge, its errors must reduce with improved knowledge.
3. To justify the cost involved in the arduous task of learning, planning with knowledge should be faster than knowledge-free planning.
4. Finally, since we are interested in planning in two-person games where there is an active opponent trying to defeat an agent's plans, it is important to account for the counter-plans of the opponent in planning.

A planning technique called Knowledge Enabled Planning (KEP) is introduced in this paper and is shown to reasonably satisfy the above requirements. Knowledge Enabled Planning is embodied in a planner which is part of a system called LEBL (Lazy Explanation-Based Learner). LEBL learns over-general macros in two-person games by generalizing incomplete explanations [14]. LEBL refines its macros when it is confronted with plan failures due to its incomplete knowledge of useful macros. The planner of LEBL composes the macros learned by the learner to plan for a given problem, and tests its plans by building a game tree consistent with the plans and counter-plans of the two players. The planner terminates when some plan of a player cannot be defeated by any opponent's plan built using the currently available macros.

The rest of the paper is organized as follows: Section 2 describes how LEBL represents its knowledge and its plans. Section 3 introduces Knowledge Enabled Planning, and illustrates it with an example from king and pawn endgames in chess. Section 4 presents some empirical results comparing our planner to another program that uses α-β search. Section 5 presents a complexity analysis of our algorithm and compares it with that of an α-β algorithm. Section 6 describes some previous work in this area and relates it to ours. Section 7 is a discussion of some of the issues involved in this kind of planning. The final section summarizes the contributions of this work.

2 Knowledge Representation

Knowledge in our system is represented by a set of inter-related goals and macros which are called *optimistic plans* or *o-plans*. A goal has a logical *definition* expressed as a quantifier-free formula, a *sign*, and a number called *promise* that indicates its relative worth. If the sign of the goal is positive, the goal is achieved when the goal definition changes its value from false to true. If the sign is negative, the goal is achieved when the value of the definition changes from true to false. The promise of a goal is used in evaluating a state (chess position). The value of a state is the sum of the promises of all different instantiations of all the goals achieved in that position.

Associated with each goal is a set of *o-plans*. The body of an o-plan is a sequence of generalized moves, each move being preceded by the weakest conditions that must be true of a state so that the rest of the move sequence is applicable in that state.[1] For example, the body of the o-plan *PL*1 (see Table 1) to queen a white pawn consists of three rules. The first rule recommends pushing a white pawn in the fifth rank if the squares in the sixth, seventh and eighth ranks in the same file are free. Similarly, the second rule says that a white pawn in the sixth rank must be pushed if the corresponding seventh and eighth rank squares are free, and so on.

Each move in the o-plan is associated with a player who makes that move. The order of moves in the o-plan is fixed, except that when two successive moves are to be made by the same player, it is assumed that there is an irrelevant move by the opponent between those two moves. Each move in the o-plan either directly achieves a part of the goal or enables another move that follows it by satisfying some of its preconditions.

Player: WHITE
Goal: QUEEN-WHITE-PAWN
Def: ((ON ?p1 ?x 8) (TYPE ?p1 PAWN) (OWNS WHITE ?p1))
Promise: 200
Body: [**If** ((ON ?p1 ?x 5) (TYPE ?p1 PAWN) (TO-PLAY WHITE) (OWNS WHITE ?p1)
(≥ ?x 1) (≤ ?x 8) (FREE ?x 6) (FREE ?x 7) (FREE ?x 8))
Then PAWN-MOVE(WHITE ?x 5 ?x 6)
If ((ON ?p1 ?x 6) (TYPE ?p1 PAWN) (TO-PLAY WHITE) (OWNS WHITE ?p1)
(≥ ?x 1) (≤ ?x 8) (FREE ?x 7) (FREE ?x 8))
Then PAWN-MOVE(WHITE ?x 6 ?x 7)
If ((ON ?p1 ?x 7) (TYPE ?p1 PAWN) (TO-PLAY WHITE) (OWNS WHITE ?p1)
(≥ ?x 1) (≤ ?x 8) (FREE ?x 8))
Then PAWN-MOVE(WHITE ?x 7 ?x 8)]
Counter-plans: [PL2, PL3]

Table 1: O-plan PL1: to queen a pawn

O-plans are related to other o-plans through *sub-plan* and *counter-plan* relations. An o-plan P is a *sub-plan* of another o-plan Q, if, under some circumstances, P *enables* some conditions necessary for Q. Similarly, an o-plan P is a *counter-plan* of Q, if, under some circumstances, achieving P *disables* some conditions necessary for Q. The o-plan *PL*1 has two counter-plans *PL*2 and *PL*3 each of which may be used to prevent the white pawn from queening by taking it.

If the preconditions of (any suffix of) an o-plan are satisfied, it is guaranteed that the goal of the o-plan

---

[1]This can be viewed as a compact way of storing a macro and all the suffixes of that macro along with the applicability conditions of each of them. As in [8], while planning, each o-plan is tried from the last suffix of its body toward the first until either the precondition of one of them matches the problem or none of the preconditions matches it. This has the advantage of trying a longer suffix of an o-plan only when no shorter suffix is applicable.

can be achieved *if* the moves in (that suffix of) that o-plan (and no other moves) are executed one after another. O-plans are optimistic since it is implicitly assumed that either the opponent's moves are irrelevant to the success of the o-plan, or they form parts of the o-plan itself. They are also approximate in that they ignore possible interactions between different o-plans or, indeed, some interactions between different moves of the same o-plan, which have not been taken into account during the learning. Due to these reasons, it is typically not possible to execute an entire o-plan successfully in a two-person game. So the planner combines o-plans into more complicated *c-plans* using *plan combinators* such as *SEQ* and *MESH*, which, among others, have the purpose of preparing for more than one alternate scenario. (Similar plan combinators are used by Bratko [2] and Campbell [3].) Trivially, every instantiated o-plan is a c-plan. Each c-plan c is associated with a search tree $T(c)$ as follows (cf. Figure 1):

$T(o\ \sigma)$	The single path defined by the move sequence of the o-plan o instantited with σ.
$T(SEQ\ p1\ p2)$	Obtained by appending each path in $T(p1)$ with each path of $T(p2)$.
$T(\ MESH\ p1\ p2)$	Obtained by interleaving each path in $T(p1)$ with each path of $T(p2)$ in all possible ways, stopping when both the paths are exhausted.

The *complexity* of a c-plan is defined as the total number of o-plans in the c-plan.

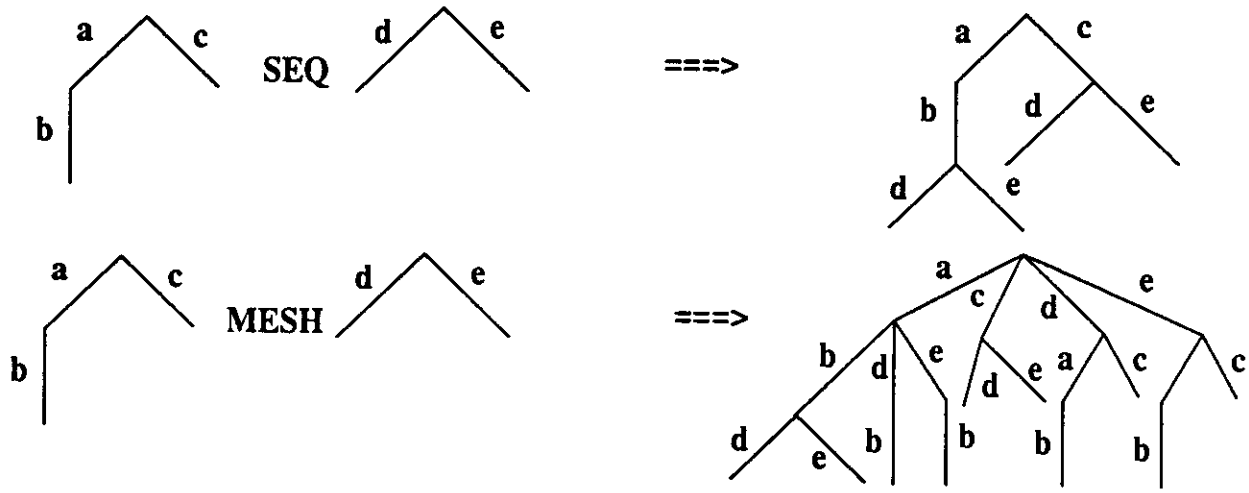

Figure 1: Plan Combinators

3 Knowledge Enabled Planning

This section describes the planning algorithm and illustrates it with an example from the king and pawn endgames.

3.1 Algorithm

The planner is given a problem state (i.e. a board position), the player for whom the system is supposed to plan, and a minimum expected evaluation for that state. It also has access to a knowledge base consisting of previously learned o-plans, and is given an upper bound k on the complexity of SEQ and MESH plans it is going to consider. The output of the planner is a partial solution tree whose min-max is at least as good as the expected value from the point of view of the player for whom it is planning, if indeed there is such a solution tree. The solution tree output by the planner is *partial* in the sense that not all possible moves of the opponent are considered at each intermediate state. Which moves do get consideration is determined by the o-plans present in the knowledge base, and the value of k. In particular, only those moves which occur in c-plans of complexity less than k constructed from the o-plans currently present in the knowledge base are considered. The minimum expected value of a position is used to prune out the plans which are not promising to meet the expectation early on, and to terminate the search as soon as the planner finds a solution tree which meets the expectation.

```
Plan (problem state S, player P, expectation E)
Initialize the old_c-plans of both the players to contain the null plan;
Initialize the game tree to S;
Initialize the c-plan pointer to the first old c-plan of P;
Loop
   Do until P succeeds or P's old c-plans are exhausted
         If there are no new_c-plans of P left
            Generate more new c-plans (of complexity < k) for P
               by composing his current old c-plan with all applicable o-plans
               increment the c-plan pointer;
         For each new c-plan N of P
            Move N to the old c-plans of P
            For the opponent's active c-plan and each of his old c-plans O
               If P is to move in S, Test(S, N, O) Else Test(S, O, N)
               {Expand the game tree by adding moves consistent with N and O;}
   End Do;
   If P succeeds exit the loop;
   Let P := opponent(P)
End Loop
Return the solution tree
End Plan

Note: P succeeds ≡ min-max(S) is better than E from P's perspective
```

Table 2: Planning Algorithm

The c-plans of each player are maintained in two lists *old_c-plans* and *new_c-plans*. *New_c-plans* are c-plans that have not yet been tested against the *old_c-plans* of the opponent. *Old_c-plans* are c-plans which have been tested and found to lose against some of the opponent's c-plans. (See Table 2 for the planning algorithm). The *old_c-plans* of each player are initialized to contain a *null plan* in the beginning. A *null plan* for a player is always applicable and simply changes the turn of the player. The purpose of a *null plan* is to simulate the effect of an irrelevant move.[2] The planner works by alternately *generating* new c-plans for each player and *testing* them against the old c-plans of the opponent. New c-plans are generated by composing the current old c-plan (pointed at by the c-plan pointer) with any o-plan using a plan combinator. Each new c-plan of a player P is tested with the old c-plans of his opponent, and, if defeated, moved to P's old c-plan list.

Testing of a c-plan against an opponent's c-plan consists of adding the move sequences consistent with both the c-plans to the existing game tree and re-evaluating its min-max. The testing algorithm makes a move according to the c-plan of the current player, and recursively calls itself on the resulting state with the remaining c-plans of the two players. (See the testing algorithm in Table 3.) After exploring all possible moves consistent with the current player's c-plan, it returns the maximum (or minimum) of the evaluations of all the children of the current state. If one player's c-plan exhausts before the other's while testing, it is assumed that the first player follows a *null plan*. A c-plan of a player P *defeats* an opponent's c-plan if it manages to keep the min-max of the game tree better than the minimum expected value (from P's perspective). If a (new) c-plan of a player defeats all the (old) c-plans of the opponent then the planner switches sides and plans for the opponent. It terminates when it fails to generate any new c-plan with a complexity less than k that can improve the min-max value for a player, and outputs a partial solution tree for the winning player -- i.e., the solution tree of the winning player when the moves of the two players are restricted to those in the explored part of the game tree.

[2] The use of a *null plan* might sometimes introduce errors in the evaluation of positions, especially when no irrelevant moves exist. Our system currently ignores the complications that arise due to this.

Test(problem state S, current player's c-plan(s) P_a, opponent's c-plan(s) P_b)
If both P_a and P_b have ended, return updated *value*(S).
Else if one of them has ended, make it a *null plan*
For all possible moves M consistent with P_a
  Do
    R_a := remaining part of P_a
    R_b := remaining part of P_b consistent with M
    S' := *apply*(S,M)
    value(S') := *Test*(S', R_b, R_a)
  End
Update *value*(S) by doing min/max on the values of children of S.
Return *value*(S)
End *Test*.

Table 3: Testing of C-plans

3.2 Example

I now illustrate Knowledge Enabled Planning in LEBL with an example from king and pawn endgames in chess. Let us assume that White is trying to maximize the value of a position, while Black is trying to minimize it. Initially, LEBL is given three goals for each player: promoting a pawn which is worth 200, (or -200 if it is a Black's pawn), taking the opponent's king of worth 10,000 (or -10,000) and taking the opponent's pawn of worth 20 (or -20). The chess problem in Figure 2 is given to LEBL after it learned a few o-plans. The system is asked to plan for White with a minimum expectation of 100. The maximum c-plan complexity is to be 3. In addition to the White's o-plan of queening the white pawn (Table 1), the portion of the knowledge base relevant to solving this problem consists of two o-plans of Black to take white pawns. One of these o-plans, *PL2*, is shown in Table 4. The body of this o-plan consists of two moves: the first move is by White and it consists of pushing a pawn $?p1$ when there is a black pawn $?p2$ two ranks ahead in its left adjacent file. The second move consists of Black's pawn $?p2$ taking the white pawn $?p1$ in its right diagonal position. Another relevant o-plan *PL3* is a symmetrical plan of Black to take white pawn from the right instead of from the left.[3]

First, the planner finds that *null plan*s by both the players do not achieve the expected min-max for White. The planner then generates all instantiations of o-plans of White applicable in that position. One of the c-plans thus generated by instantiating the o-plan *PL1* is to queen the *c5* pawn of White. This c-plan is then tested against Black's *null plan*, and is found to win, i.e. change the min-max value of the initial state to 200, which is more than the minimum expected value. The planner then switches sides and generates a new c-plan for Black by instantiating *PL2*: the plan to take the white pawn which is moved to *c6* with the black pawn at *b7*. This c-plan of Black is found to succeed against the White's c-plan reducing the min-max value to -20. The planner again switches sides and tries the remaining new c-plan of White which is still not tested: pushing the *a5* pawn to queen. This is tested against the two old c-plans of Black, and is found to succeed. Once again, by instantiating *PL3*, the planner generates a new c-plan of taking the queening white pawn with the black pawn at *b7*, when it is on *a6*. This c-plan is found to successfully defeat White's new c-plan.

[3]The system learns two different o-plans for these two kinds of capture simply because the domain theory and the operational predicates are sensitive to the direction. This is not a limitation of the planner, but an effect of our representation choice on the explanation-based learner.

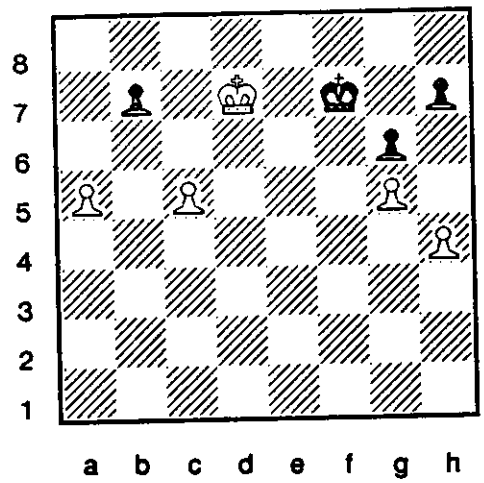

Figure 2: White to Play

Player: BLACK
Goal: TAKE-WHITE-PAWN
Def: ~((OWNS WHITE ?p1) (TYPE ?p1 PAWN))
Promise: -20
Body: [**If** ((ON ?p2 ?X1 ?Y1) (ON ?p1 ?X ?Y) (TYPE ?p2 PAWN) (TYPE ?p1 PAWN)
(OWNS BLACK ?p2) (OWNS WHITE ?p1) (PLUS1 ?X1 ?X) (PLUS1 ?Y2 ?Y1)
(PLUS1 ?Y ?Y2) (≤ ?Y2 8) (≤ ?X 8) (≥ ?Y2 1)
(≥ ?X 1) (FREE ?X ?Y2))
Then (PAWN-MOVE WHITE ?X ?Y ?X ?Y2)
If ((ON ?p2 ?X1 ?Y1) (ON ?p1 ?X ?Y2) (TYPE ?p2 PAWN) (TYPE ?p1 PAWN)
(OWNS BLACK ?p2) (OWNS WHITE ?p1) (PLUS1 ?X1 ?X) (PLUS1 ?Y2 ?Y1))
Then (PAWN-TAKE BLACK ?X1 ?Y1 ?X ?Y2)]
Counter-plan-of: [PL1]

Table 4: O-plan PL2: To take White's pawn

Using the MESH combinator, the planner then generates the c-plan of interleaving the two White's c-plans, i.e. pushing both *a5* and *c5* pawns in all possible orders. This c-plan is tested against all the old c-plans of Black and is found to succeed with a final evaluation of 180. Black then generates some more c-plans such as interleaving its old c-plans, but none of them are successful against the White's c-plan. After exploring all such c-plans of Black of complexity $\leq k$, and finding that none of them works, the planner outputs the current solution tree for White for that position. See Figure 3 for the game tree explored and the solution tree output by the planner.

Notice that only a small percentage of all possible moves of both the players are explored. In particular, no moves have been explored without a *purpose*, i.e. without being part of a specific c-plan to achieve some goal. Hence, only a few directed moves of the black and white pawns are explored, and none of the king moves are explored. The system does not replan *during* the course of testing a given pair of c-plans, which also helps reduce the search considerably. While this approach might seem too weak and fallible for a complex domain like chess, incremental learning of o-plans from plan failures makes it self-correcting and practical.

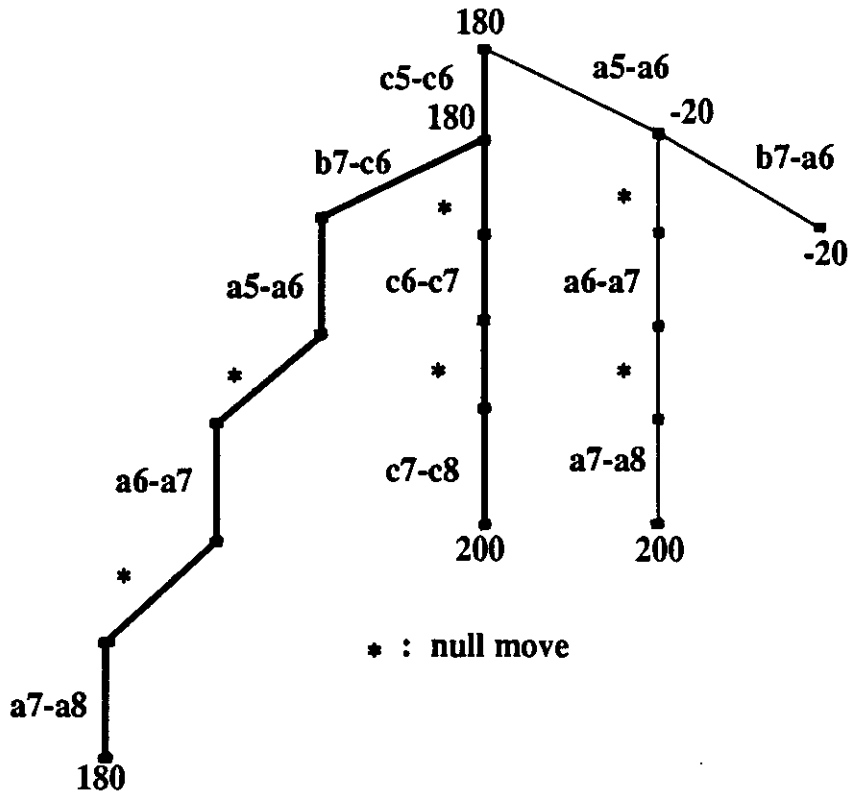

Figure 3: A part of the search tree explored for the example.
The thicker lines indicate the partial solution tree output by the planner.

3.3 Optimizations

A straight-forward implementation of the planning algorithm can be quite inefficient. Several optimizations are used to make it efficient.

1. O-plan relevance: Every o-plan in the two c-plans is checked to see if it is *relevant.* An o-plan is relevant if (a) it has a non-zero promise, or (b) it is a sub-plan of a relevant o-plan in the same player's c-plan, or (c) it is a counter-plan of an o-plan in the opponent's active c-plans. If some o-plan in either of the two c-plans is not relevant, then the c-plans are not tested against each other.

2. Move set hashing: While testing a MESH plan, adding the moves consistent with the c-plan to the game tree requires prohibitively expensive search, since that involves exploring all possible orderings of two or more o-plans (see next section for details). However, we can design a faster algorithm by making some assumptions about the representation of operators. We assume that the bindings of the operator preconditions completely determine the differences between the initial and the resultant states of the operator application. We call such operators *tight* operators.[4] If all the operators in the domain are tight, it can be shown that any two move sequences made up of the same set of instantiated operators, when played from the same state, result in the same final state. For example, the move sequence $<m1, o1, m2,$

[4]An example of an operator which is not tight is TOGGLE; it is always applicable, and if the switch is initially ON, it turns it OFF and vice-versa. Another example may be CLEAR-TABLE, which is also always applicable, and clears all the blocks on the table. In each case, the preconditions do not bind the variables to specific parameters which are going to determine the exact effects of the operators.

$o2$> and the move sequence <$m2$, $o2$, $m1$, $o1$>, when played from the same state result in the same final state.[5] This property is exploited by hashing a state by the set of moves ({ $m1$, $m2$, $o1$, $o2$} in our example) in the move sequence used to reach that state.

3. Within-plan α-β: In ordinary min-max search α-β cutoffs preserve the correctness of the solution. We can not directly incorporate α-β cutoffs in our system because its correctness preserving property depends on the fact that the game tree is explored depth first. LEBL explores the game tree in depth first order only during the testing phase. Hence, we use α-β cutoffs during the testing of each pair of c-plans.

4. C-plan subsumption: Some c-plans are subsumed by other c-plans in the sense that the game tree that corresponds to one is guaranteed to be a subtree of the game tree that corresponds to the other. E.g., <*MESH* O_1 O_2> subsumes <*SEQ* O_1 O_2>. If two old c-plans O_p and O_q of the two players p and q which have been tested against each other respectively subsume the two new c-plans N_p and N_q, then the new c-plans need not be tested. This results in large savings if it is applied at all nodes in the game tree by caching the latest two c-plans which have been tested against each other at that node and skipping the testing of any two new c-plans subsumed by the cached c-plans.

The efficiency of LEBL significantly increased due to these optimizations. Unless otherwise specified, "LEBL" corresponds to a program with all the above optimizations.

4 Empirical Results

This section demonstrates two things: first, it shows that Knowledge Enabled Planning involves significantly less search than knowledge-free techniques like α-β search in our domain, king and pawn endgames. Second, it demonstrates that the errors, defined as the incorrect first moves on test problems, reduce in our method with increased knowledge.

4.1 Comparison of Search Effort

To compare the search effort of LEBL with that of a knowledge-free planner, a program called ABE which is based on α-β search is constructed. ABE accepts a board position, the minimum expected evaluation for the state, and the maximum depth of search as inputs, builds a complete solution tree for that position using the α-β algorithm, and outputs the number of nodes searched. ABE uses the same evaluation function as LEBL, i.e. the sum of the promises of all goals achieved in a given position. ABE is also given the same minimum expected evaluation of the state as LEBL. The maximum height of the correct solution trees for the problems is input to ABE as the maximum depth of search. This is probably unfairly favorable to ABE since one would typically not know the height of the correct solution tree and LEBL is not given this information. However, except for the o-plans, LEBL does not have any more information than ABE.

A set of 19 examples was selected from the king and pawn endgames to train LEBL on, and 8 completely different problems (with known solutions) were chosen to test it on. The column labeled "ABE" in Table 5 shows the number of nodes visited by ABE on the 8 test problems. The next three columns show the number of nodes visited by LEBL on the test problems after training it on the first 6, 12 and all 19 of the training examples respectively. In this training session, LEBL learned a total of 25 o-plans and 5 new goals. After finishing one training session, LEBL is again trained on the same 19 training examples. It learned a total of 5 new o-plans for the 5 new goals it learned in the first session. It is then tested on the 8 test problems once again. The number of nodes visited in this session are shown in the last column of Table 5.

The results shown in Table 5 indicate that LEBL's planner consistently searched one to two orders of magnitude fewer nodes than ABE. Though the number of nodes visited by LEBL usually increased with the number of o-plans, it still remained significantly less than the number visited by ABE. Surprisingly, in a

[5]Some of these sequences might not be legal in a given position. However, they reach the same final state when they are legal.

#	ABE	LEBL (6)	LEBL (12)	LEBL (19)	LEBL (2X19)
1	>50,000	10	878	1,374	294
2	>50,000	10	1,199	2,017	2,017
3	>50,000	14	97	352	68
4	>50,000	17	33	1,294	1,150
5	157	6	18	33	33
6	1,361	7	21	21	21
7	47,888	14	92	180	14
8	971	5	5	9	9

Table 5: The number of nodes visited: ABE vs LEBL

few cases, the amount of search *decreased* with learning more o-plans. The reason for this is that when there is no o-plan present in the system to achieve a goal, it tries to combine a number of o-plans to achieve it. This causes additional search which is avoided by having an o-plan which directly achieves the goal.

4.2 Variation of Errors

While the amount of search in LEBL was much less than ABE, it made errors while ABE did not. However, as shown below, the number of errors made by LEBL decreased with increaed knowledge. The errors may be measured either by the number of incorrect first moves on a problem, or the number of incorrect evaluations of min-max on the 8 test problems. Both these measures are reported in the table below.

# training examples	# o-plans learned	# incorrect first moves	# incorrect evaluations
0	0	8	8
6	8	2	5
12	17	0	5
19	25	0	3
2X19	30	0	3

Table 6: Variation of errors with knowledge

The first column of Table 6 indicates the number of examples LEBL was trained on, the second column shows the total number of o-plans in the system after the training, the third column shows the number of test problems on which LEBL's first move was incorrect, and the fourth column shows the number of problems for which LEBL's evaluation (minmax) was incorrect. It is clear from the table that both the number of incorrect evaluations and the number of incorrect first moves in LEBL decreased gradually with learning.

Though the above results appear good, the sample is too small to make any general conclusions, and it is to be seen whether this approach scales up sufficiently well with the number of o-plans in the system. In other words, we need to study the asymptotic behavior of our system more thoroughly to discover its strengths and limitations.

5 Complexity Analysis

The empirical results are sensitive to many domain parameters such as how the domain is represented, how many o-plans are learned and how good the learned o-plans are. Unless one does a thorough empirical investigation varying all these parameters systematically, it is difficult to draw any definite conclusions from empirical studies alone. In this section, we turn to complexity analysis of the algorithms used by the two programs to complement the knowledge gained through empirical studies.

We first estimate the complexity of planning in LEBL. In our analysis, we assume that the c-plan complexity is bounded by k. We let n be the *average o-plan branching factor*, i.e. the average number of o-plan matches in each state, and let l be the average length of an o-plan. We make the assumption that the cost of matching an operator precondition to a state description and the cost of matching an o-plan precondition to the state description are of the same order, and denote this cost by *Match_cost*.[6] We let p denote the number of o-plans in the knowledge base.

We estimate the planning complexity by dividing it into its two components, *generation* and *testing*. In the worst case, the planner generates all possible c-plans of complexity $\leq k$ for each player. Hence, the maximum number of possible c-plans generated for each player is $(2n)^k$. The factor 2 in the base here is due to the number of plan combinators (SEQ and MESH), and n is the o-plan branching factor.

In the worst case, a match is attempted for every suffix of every o-plan, whenever new c-plans are generated. Hence the total match cost during the generation of c-plans for each player is:

$$(2n)^k pl \cdot Match\_cost.$$

Each c-plan of each player is tested against each c-plan of the opponent. The testing of two c-plans in the worst case involves generating all possible ways of interleaving all the o-plans in the two c-plans. Doing this in the straight forward way involves generating $(2kl)!/(l!)^{2k}$ states in the worst case. That is indeed too many states! However, assuming that all our operators are tight, and using the hashing scheme described earlier reduces the complexity to $k \cdot l^{2k}$. The term l^{2k} is the number of different intermediate states when interleaving k o-plans of each player. The additional factor k occurs since each such state can be visited at most k times, each time by advancing a different o-plan of the same player.

So the complexity of testing all c-plans that have at most k o-plans for *each* player is given by the following:

$$(2n)^{2k} \cdot k \cdot l^{2k}$$

Adding the match cost to the above, and simplifying, we have

$$\text{Planning cost} := O((2nl)^{2k} k \cdot pl \cdot Match\_cost)$$

Let us now turn to the worst case complexity[7] of the α-β algorithm. In order to be able to always find the solution that LEBL's planner finds, the average depth to which the α-β algorithm must search is $2lk$. If the average branching factor of the game tree is b, we have

$$\text{Complexity of } \alpha\text{-}\beta := O(b^{2lk} \cdot Match\_cost)$$

Let us compare the above to the complexity of KEP. Since we assumed that the match costs in both the algorithms are of the same order of complexity, the major savings in KEP is going to come from the exponent. Since the complexity of our method has only the exponent $2k$, compared to $2lk$ of α-β, we can say that KEP performs better if the average length of the o-plans l is high. Further, it helps if the total number of o-plans p, the average o-plan branching factor n, and the maximum allowed plan complexity k

---

[6] This assumption is not true if the preconditions are general open conjunctive formulae, since the match problem for them is exponential in the length of the formulae. However, sorting the terms in our preconditions in a predefined order of predicates seems to keep the match cost in our domain sufficiently low so that this assumption is not unreasonable.

[7] We expect that a comparision of corresponding average complexities would be similar.

are low. In other words, our analysis reveals that Knowledge Enabled Planning out-performs α-β algorithm if the natural distribution of problems in a domain is such that a *high proportion* of the problems are solvable by c-plans of *low* complexity built from a *small* library of *long* o-plans with a *small* branching factor.

6 Related Work

There are many planning systems that plan using learned macros (see [8], and [10] for example). However, all these systems assume that the learned knowledge is guaranteed to be correct since it is learned by generalizing proofs formed of truth preserving inference rules. These planning systems are not directly usable in highly combinatorial domains where the learned knowledge is usually only approximately correct.

Carbonell, in his POLITICS system, addressed the issue of adversary planning in natural language comprehension [4]. His system contained several heuristic strategies applicable to general conflict situations such as causing resource conflicts, goal compromise etc., and a control algorithm that determines when to apply them. However, the control algorithm, and the heuristic strategies of POLITICS are quite involved, and more importantly, the notion of time is not adequately represented which makes it unsuitable for game domains.

PARADISE, a program developed by David Wilkins, used chess knowledge encoded as production rules to plan in tactical middle game positions [16]. PARADISE used chess knowledge to selectively search the game tree. However, the knowledge of PARADISE is hand-coded and domain-specific. It uses high level constructs like safely-capture-a-piece, safely-move-a-piece etc. which are implemented using production rules. Until we know how to learn these high level concepts automatically, this approach to planning is difficult to integrate with learning.

Bratko developed a program called AL3 which was used to plan in KPK (King and Pawn vs. King) endings [2] in chess. While AL3 and LEBL have many similarities and are used on similar problems, there are some important differences. One difference between AL3 and LEBL is that chess knowledge in AL3 is hand-coded rather than learned. Another difference is that LEBL's planning algorithm uses min-max to evaluate a position and has a numerical measure to denote the worth of the goals. AL3, on the other hand, measures success depending on the achievement of goals. AL3 expressed chess knowledge as constraints on possible moves to achieve new goals while protecting the old goals, where as knowledge is represented in LEBL as inter-related goals and o-plans. Finally, the AL3 planner terminates after searching a predetermined maximum number of nodes. The termination .condition for our planner depends only on the availability of relevant plan knowledge and the maximum allowed c-plan complexity.

Another program which solves problems in king and pawn endgames in chess is CHUNKER developed by Murray Campbell [3]. CHUNKER exploits the fact that the pawns and kings in a chess endgame position can be aggregated into small groups called *chunks* such that the interactions between different chunks are minimal. The planner works by planning for each chunk separately, combining the plans, and checking for possible interactions between the plans. It appears that the control of o-plan use can be improved by methods like this if we are able to generalize the notion of chunks so that it is applicable to other domains.

7 Discussion

The need to plan with incomplete or incorrect knowledge has not been addressed until recently. However, in many realistic domains, acquisition of complete and correct knowledge is intractable, or even impossible. In this paper, we discussed a planner that effectively uses the over-general o-plans learned through Lazy Explanation-Based Learning to plan in game domains. It may be argued that if the learner learned general c-plans instead of o-plans, a complicated planner like ours wouldn't be necessary. A simple instantiator of c-plans would suffice. However, the validity of this line of argument rests on how frequently each *c-plan* occurs as a solution to the problems in the domain. If the frequency of occurrence of each generalized c-plan is low, then it makes more sense to compose c-plans out of smaller

components like o-plans at the time of the problem-solving than to explicitly learn c-plans. If each c-plan repeats with high frequency, then it is better to explicitly learn them. It is our belief that in our domain the frequency of occurrence of o-plans is high while the frequency of occurrence of c-plans is low. Hence, it is better to explicitly learn o-plans and compose c-plans at the problem solving time.[8]However, this statement needs to be supported with more empirical evidence for the effectiveness of o-plans to represent planning knowledge in many domains.

This raises another interesting question: why should the system learn o-plans instead of planning from the scratch every time it sees a problem? Surely, the frequency of occurrence of primitive moves in a solution is higher than that of any set of o-plans. However, as we have shown in our complexity analysis, the higher the grain-size of the primitives with which planning is done, i.e., the longer the o-plans, the lower the complexity of planning. So it makes sense to choose units of larger grain size, if they are frequent enough in the solutions of problems we are interested in solving. This suggests the following solution to the choice of grain size problem [11]: choose the largest grain-size at which the primitives are sufficiently frequent in the solutions of problems of interest. It might be possible to quantify this insight more precisely in the future.

Our system also illustrates a fundamental tradeoff between learning time and problem-solving or planning time. A system that learns c-plans explicitly, and simply instantiates them during planning requires learning every plan that occurs in the solutions of problems of interest. Hence, it needs more learning time, and less planning time. A system that plans from scratch for every problem requires no learning time but takes very long to plan. Our system strikes a compromise between these two extremes. It would be interesting to see more systems as data points on this spectrum.

It might appear paradoxical that the number of nodes searched by our planner should *increase* with the number of learned o-plans. However, it must be remembered that limited knowledge leads to erroneous plans in our system especially when those limitations are relevant to the problem at hand. As the system learns more o-plans, its search becomes more exhaustive, and the number of errors in its planning decrease. It may also be necessary to learn higher level control knowledge about using c-plans instead of trying to use all applicable c-plans one after another. There are two sources of high complexity in our planning. The first one is due to lack of sufficient guidance in plan generation. For example, LEBL proposes to compose any two o-plans by SEQ operator, if it happens to see one instance in which they are successfully composed by SEQ. This is because it does not explicitly learn the conditions under which such composition is going to succeed, and test them before composing. On the other hand, explicitly learning such conditions might be too expensive, since there may be too many such conditions. The other source of high complexity in planning is while testing c-plans. Our testing algorithm is too conservative in the sense that it assumes the worst possible interactions between the o-plans. A better approach might be to assume independence of o-plans and learn about interactions when the c-plans fail. The planner might check for only explicitly learned interactions during the planning. This approach works better if the solution space is such that the o-plans interact minimally.

Another source of high complexity in planning is the match cost of o-plans. In the logic based representation we currently use, matching of open conjunctive formulae is known to be NP-hard. One way to tackle this problem is to reduce the expressiveness of the language of o-plan preconditions. A similar solution is proposed to avoid expensive chunks in SOAR by Tambe and Rosenbloom [15]. Though this limits the kinds of o-plans we could express in our system, we think that most natural domains allow languages much less expressive than open conjunctive formulae to represent their plan preconditions.

A complimentary problem is the lack of adequate expressive power in the action part of the o-plans and c-plans. Currently o-plans are limited to causally connected sequences of moves. It sometimes requires too many such o-plans to express some piece of useful domain knowledge, which may be more compactly represented as a loop or a sequence of high level actions. Learning and planning with such

---

[8]It is claimed that a similar property underlies the reason behind the learning curve exhibited by SOAR. Learning in the beginning, when the chunks formed are of smaller size, is faster since the frequency of occurrence of smaller chunks in the solution space is greater [9].

complicated o-plans is non-trivial. For example, learning macros with loops has been a difficult problem in EBL, though there have been a number of solutions proposed recently [12, 13, 6]. To plan with o-plans involving abstract actions is also difficult, since it would probably require base level search to instantiate the abstract actions with primitive moves. We need to study the relation between the expressiveness of the plan languages and the complexities of planning and learning in those languages. This seems to us to be the central issue in learning plan knowledge and using it to solve problems.

8 Conclusion

In this paper, a knowledge based planning technique called Knowledge Enabled Planning (KEP) is introduced using a program called LEBL that makes use of approximately learned planning knowledge to plan effectively in game domains. LEBL is found to satisfy several properties desirable in such a knowledge based planner. In particular, (a) LEBL accounts for imperfections in o-plans by composing them and checking for interactions (b) it makes less errors with increased knowledge by searching more exhaustively (c) it is faster than knowledge-free planning in some domains, and (d) it takes into account the presence of an active adversary. The above claims are empirically supported by our program in the domain of king and pawn endgames in chess. A complexity analysis of our planning algorithm is presented and is used to derive the conditions under which Knowledge Enabled Planning performs better than α-β search under the worst case assumptions. Several questions such as the asymptotic behavior of Knowledge Enabled Planning, the adequacy of o-plans to represent planning knowledge, and the tradeoffs between planning and learning costs in systems like ours remain to be answered.

Acknowledgments

I am indebted to my advisor, Tom Mitchell, for his constant support, many illuminating discussions, and encouragement throughout this work. Thanks are also due to Jack Mostow for encouraging me to work on this problem, and for discussing with me many of the ideas presented here. I thank Steve Chien, Oren Etzioni, Tom Mitchell, Jack Mostow, and Ming Tan for many useful comments on an earlier draft of this report.

References

[1] Bennett, S.
Approximation in Mathematical Domains.
In *Proceedings IJCAI-10.* Milan, Italy, 1987.

[2] Bratko, I.
Advice and Planning in Chess Endgames.
In *Artificial and Human Intelligence.* Elsevier Science Publishers, B. V., 1984.

[3] Campbell, M.
Chunking as an Abstraction Mechanism.
PhD thesis, School of Computer Science, Carnegie Mellon University, 1988.

[4] Carbonell, J. G.
Counterplanning: A Strategy-Based Model of Adversary Planning in Real-World Situations.
Artificial Intelligence 16, 1981.

[5] Chien, S.,
Simplifications in Temporal Persistence: An Approach to the Intractable Domain Theory Problem in Explanation-Based Learning.
Technical Report UILU-ENG-87-2255, University of Illinois, 1987.

[6] Cohen, W.,
Generalizing Number and Learning from Multiple Examples in Explanation Based Learning.
In *Proceedings AAAI-88.* St. Paul, Minnesota, 1988.

[7] Ellman, T.,
Explanation-Directed Search for Simplifying Assumptions.
In *Proceedings of Spring Symposium Series: Explanation-Based Learning.* 1988.

[8] Fikes, R. E., Hart, P. E. and Nilsson, N. J.
Learning and Executing Generalized Robot Plans.
Artificial Intelligence 3(4):251-288, Winter, 1972.

[9] Laird, J. E. and Rosenbloom, P. S.
Toward Chunking as a General Learning Mechanism.
In *Proceedings AAAI-84*, pages 188-192. Austin, TX, August, 1984.

[10] Minton, S.
Constraint-Based Generalization: Learning Game-Playing Plans From Single Examples.
In *Proceedings AAAI-84*, pages 251-254. Austin, TX, August, 1984.

[11] Mitchell, T. M., Mahadevan, S. and Steinberg, L.
LEAP: A Learning Apprentice for VLSI Design.
In *Proceedings IJCAI-9*. Los Angeles, CA, August, 1985.

[12] Prieditis, A. E.
Discovery of Algorithms from Weak Methods.
In *Proceedings of the International Meeting on Advances in Learning*. Les Arcs, Switzerland, 1986.

[13] Shavlik, J., and DeJong, J.
BAGGER: An EBL System that Extends and Generalizes Explanations.
In *Proceedings AAAI-87*. Seattle, WA, July, 1987.

[14] Tadepalli, P.
Lazy Explantion-Based Learning: A Solution to the Intractable Theory Problem.
In *Proceedings of IJCAI*. 1989.

[15] Tambe, M., Rosenbloom, P.
Eliminating Expensive Chunks.
Technical Report CMU-CS-88-189, Carnegie-Mellon University, 1988.

[16] Wilkins, D.,
Patterns and Plans in Chess.
Artificial Intelligence 14, 1980.