5-2007

Soar Checkers - An Intelligent Checkers Playing Agent in Soar

Jithu Menon
Clemson University, jmenon@clemson.edu

SOAR CHECKERS - AN INTELLIGENT CHECKERS PLAYING AGENT IN SOAR

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Electrical Engineering

---

by
Jithu Menon
May 2007

Accepted by:
Dr. Robert Schalkoff, Committee Chair
Dr. John Gowdy
Dr. Stan Birchfield

ABSTRACT

The classic board game of checkers is ideally suited for research in AI game-playing programs or agents. The objective behind Soar Checkers is to investigate if it is possible to create an agent-based game playing system that would beat novices with ease and at least challenge advanced novice to intermediate-level players by designing a rules-based expert system whose knowledge base consists of nothing more than the rules of checkers, and a set of guidelines for game-play based on good strategy. Soar was chosen as the platform for building this agent because it came built-in with features that facilitate creating rules-based expert systems, has been proven to be fairly reliable in developing such systems (including flight simulators) for over twenty years and makes it relatively straight-forward to have multiple agents play each other, and to add or modify features or strategies to the agents. Though the problem definition makes it inherently hard to objectively quantify the results, the objectives were successfully achieved for the most part. It was also seen that all other things being equal, the player going second (White) has a built-in advantage, thereby confirming a widely held belief among the checkers community.

DEDICATION

To my parents Gopal and Jaya, and my brother Rahul.

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support and help of a bunch of great people. First and foremost, I would like to thank my advisor Dr. Schalkoff, who came up with the initial idea for this project and with his timely and truly helpful advice, proved instrumental in ensuring its successful completion. I would also like to thank Dr. Birchfield and Dr. Gowdy for serving on my committee and for taking time out of their busy schedules to aid me in creating the final draft.

There were numerous occasions during the course of this project when I found the going really tough especially since I was a complete Soar newbie. I would like to express my heartfelt gratitude to Mr. Bob Marinier and Ms. Karen Coulter at the University of Michigan, and Mr. Doug Pearson at Three Penny Software Inc. for their infinite patience and invaluable help whenever I had difficulties with Soar.

Last but certainly not the least; I would like to thank my family and friends for their constant support and encouragement. Special thanks to Rob Morefield for helping me keep my head up, having faith in me and for being a real friend. I would like to thank Sunil Guduru, for helping me stay on track and Dr. Bill Bauerle for his support. Thanks to everyone who wished me well - I deeply appreciate it.

TABLE OF CONTENTS

Table of Contents (Continued)

Page

Table of Contents (Continued)

Page

Table of Contents (Continued)

Page

LIST OF TABLES

LIST OF FIGURES

List of Figures (Continued)

CHAPTER ONE
PROBLEM DESCRIPTION

1.1 Introduction

The classic board game of checkers (also referred to as 8 x 8 draughts in Britain)[1] is ideally suited for research in AI game-playing programs or agents. Its similarity to chess makes it possible to leverage the advances made and lessons learnt while developing chess playing super computers such as Deep Blue and Fritz. The reduced complexity of checkers allows the researcher more leeway in exploring new techniques [1].

The objective behind Soar Checkers is to investigate if it is possible to create a game playing system that would beat novices with ease and at least challenge advanced novice to intermediate-level players by designing a rules-based expert system whose knowledge base consists of nothing more than the rules of checkers, and a set of guidelines for game-play based on good strategy. This is in contrast to the traditional AI techniques used by programs such as Chinook, which largely rely on treating game-play as a problem of search. Thus, the main problem being approached and investigated is to develop an intelligent agent in Soar[2] that

- Is aware of the rules of checkers;
- Plays at a respectable level; i.e., plays at the level of an advanced novice to intermediate level player;
- Does not use the traditional AI techniques that have been used in the past to develop game-playing programs, as described in Section 1.2 on the next page and

[1] For more information on the history and rules of checkers, please refer to Appendix A
[2] To learn more about Soar, please refer to Appendix B.

- Provides a platform for exploring the effects of various strategies and tactics on improving game play.

1.2 Games and Search

1.2.1 Motivation for Search

Most games can be represented in terms of an initial state, intermediate states, and (possibly) a (set of) final state(s). For example, in the case of chess, the initial state would be the starting board position, the intermediate states would be the board positions resulting from various moves made by the players as the game progresses, and the final state(s) would be board positions in which one player has emerged as the clear victor (for instance, a checkmate), or the game is drawn. Therefore, these states form a state-space. Evidently, for any one player, some of the final states are more desirable than the others. It could then be seen that games could be approached as a quest to traverse this state-space, starting at the initial state and ending in one of the desirable final states (for instance, win or draw). State-space search has been used quite successfully in the past to create game-playing programs. The following sections describe some of the commonly used techniques for building such systems.

1.2.2 The Minimax Algorithm

1.2.2.1 Zero-Sum Game

A zero-sum game is one in which the amount of resources available to the players involved (usually, two or more) is fixed [2]. This means that a gain for one player would result in an equal loss for the other players. In other words, if the gains and losses of all

players playing a zero-sum game are added up, the result would be zero. Examples include chess, go, checkers etc.

1.2.2.2 Introduction to Minimax Algorithm

Almost all "intelligent" game-playing computers generally use some variation or the other of the basic minimax algorithm. This is because almost all of them treat games as a problem of state-space search. In this case, the program tries to identify various paths through the state-space leading to desirable final states, and then tries to pick moves that will still lead to a desirable outcome, while expecting the opponent to do his or her best to thwart these moves. It can be seen that an algorithm that performs this state-space search would be at the heart of such search-based game-playing systems. The minimax algorithm basically recommends the best possible legal move in two-player, zero-sum games [3]. This algorithm was first proposed in 1928 by Jon Von Neumann.

1.2.2.3 Evaluation Functions

The algorithm requires an evaluation function that would assign a score to each game state (for example, board position in board games such as chess or checkers). Typically, these functions are referred to as position evaluation functions. A "good"' evaluation function is quite often the difference between various competing game-playing programs. In the case of checkers, an example evaluation function could assign different scores for the two kinds of pieces (example, 1 for an ordinary checker, and 2 for a king, various squares on the board could be differentiated using scores assigned to them (example, 5 for center squares occupied by the players' pieces, 2 for edge squares, -3 for each square near the players' king row that is occupied by an opponent checker, etc.). This evaluation function would therefore

be a weighted sum of various factors, with the weights picked to reflect both material strength and positional strength (example, a board position with positional score of 22 and having 4 checkers and 2 kings would result in a total score of 22 + (4 x 1) + (2 x 2) = 30) . The algorithm works roughly as follows [3, 4, 5]:

1.2.2.4 Components of the Algorithm

Consider two players, MAX and MIN playing a zero-sum game. The objective of MAX is to maximize the minimum possible score returned by the evaluation function used by the minimax algorithm, while that of MIN is to minimize the maximum possible score returned by the evaluation function. Suppose it is MAX's turn to make a move. The current game state is passed to a function that would first generate all the possible legal moves that MAX could make, and then return the collection (possibly zero or more) of game states that are the result of making those moves. The resulting game states are at what is referred to as the "min-level", and the algorithm works recursively, treating each of the resulting game states as the current state, generating the possible legal moves and the new game states that result after each of those moves are made - which would now be at the "max-level".

1.2.2.5 Game-Tree

A game-tree is thus generated, with the root being the current game state, the children being the game states that result from making a legal move. In the ideal case, the algorithm would run until one of the possible final game states is generated (for example, one of the players win) - however, in most cases, this would require an unacceptably long period of time, even with the most modern super computers. Hence, depending on the platform running the algorithm, the maximum search depth is limited, to say, "d".

1.2.2.6 Evaluation of Game-Tree

Once all the game states have been generated up to depth d, each of the leaf nodes is assigned the score returned by the evaluation function. If the leaves happen to be at the max-level, their immediate parents are at the min-level, and each parent would be assigned their children's minimum score. On the other hand, if the leaves are at the min-level, their parents would be assigned their children's maximum score. This process continues recursively, in a bottom-up fashion until the immediate children of the root node have been assigned scores. Now, assuming that it is the MAX players' turn to move, the move that would result in the game state with the highest score that is also an immediate child of the root would be returned as the "best" possible move that the player could make, by the algorithm. The algorithm can be described, in pseudo-code fashion as shown in Algorithm 1.1 on the next page [4, 5].

1.2.2.7 An Example

An example game-tree generated by the minimax algorithm is shown in Figure 1.1 on page 7. For the sake of clarity and brevity, we assume that the algorithm looks ahead only 3 plys[3] and that the player who is about to make the move tries to maximize the score returned by the evaluation function (thereby, becoming "MAX") and his or her opponent tries to minimize the same (thereby, becoming "MIN"). A closer inspection of the game tree generated by the algorithm reveals that from the present state of the game, MAX has two possible choices, each of which provides MIN with two possible responses. The position evaluation function is applied to the game states at the leaf nodes, and their immediate parents are assigned the highest score among their children, since the parents are at a MAX

[3] In game-playing parlance, a ply is a half-move. When both players have each made a move, it is two plys.

level. The scores are thus backed up, until the root node is reached, and in this specific case, the minimax algorithm would recommend that MAX pick the move which results in a score of 5 over one that results in a score of 3.

```
function minimax(node, depth)
  if node is a terminal node or depth = 0
      return the heuristic value of node
  if the adversary is to play at node
      let α := +∞
      foreach child of node
          α := min(α, minimax(child, depth-1))
  else {we are to play at node}
      let α := -∞
      foreach child of node
          α := max(α, minimax(child, depth-1))
  return α
```

Algorithm 1.1: Pseudo-Code for the Minimax Algorithm

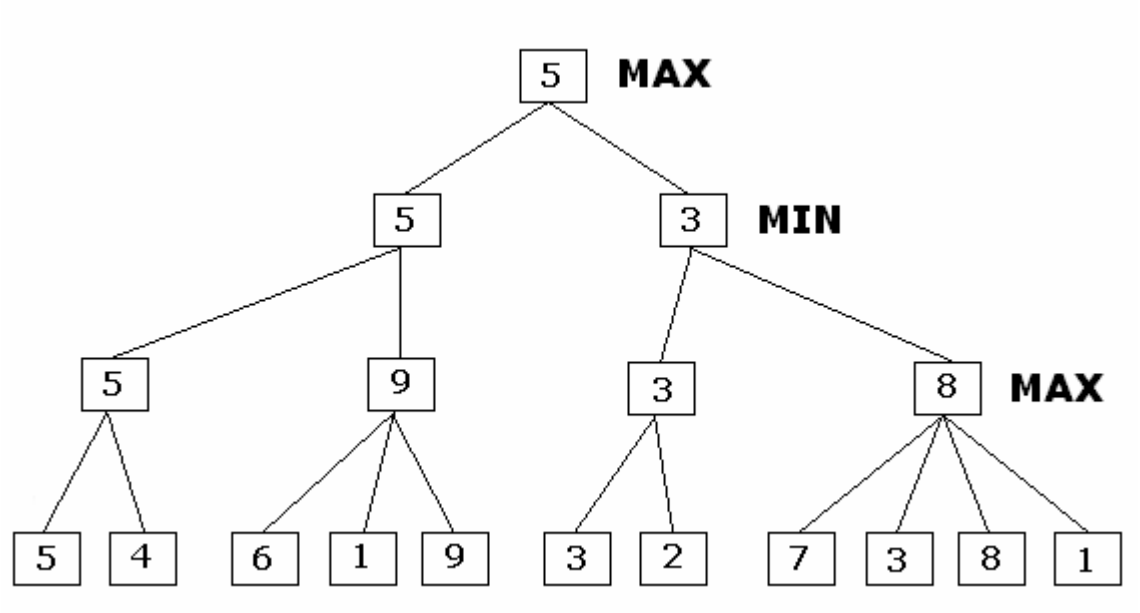

Figure 1.1: Example of a Minimax Game Tree

1.2.3 Minimax Improved: Alpha-Beta Pruning

1.2.3.1 Motivation for Improving Minimax

The major drawback with the minimax algorithm is that, though it can ostensibly predict the "best" possible move available to a player, given a certain state of the game (assuming that the position evaluation functions used are reasonably good), as the branching factor (number of legal moves possible from a given game state) increases, the search-depth, and thus the time required to perform the search for the best move increases exponentially. For example, suppose we are using the minimax algorithm to predict the best moves for a game which has an average branching factor of 10. This means that if our "intelligent" game-player looks ahead one ply, there would be 10 leaf nodes in the minimax game tree, and the depth would be 1. If it looks ahead two plys (i.e. one complete move by both players), the game tree would have a depth of 2, and at the leaf level, there would be 100 nodes. This can be summarized briefly as shown in Table 1.1 on the next page.

The scenario shown above is quite realistic in the case of checkers, since for non-capture moves, the average branching factor in checkers is 8 and for capture moves, it is approximately 1.25 [1]. It is quite common for expert players to look ahead 20 plys or more, especially in the end game. The number of leaf nodes that would have to be searched for different plys involving non-capture moves can be seen in Table 1.2 on page 9.

It is quite obvious that even with the fastest supercomputers, the time required to perform a 20 ply search would be quite unwieldy. Significant gains can be made in the time required to generate the best move, if the number of nodes involved in the search can be drastically reduced. This is where a technique called Alpha-Beta pruning [3, 6], and its variants come into play. It is safe to say that almost all search-based game playing programs rely on minimax search with alpha-beta pruning for their "intelligent" game-play.

Look-Ahead Depth	Number of Leaf Nodes
1	10
2	100
3	1000
4	10000
5	100000
6	1000000
.	.
20	100000000000000000000

Table 1.1: An Example of Exponential Increase in Number of Leaf Nodes to be Searched with Increasing Depth of Search when the Branching Factor is 10

1.2.3.2 An Example

Let us consider the same example that was used to explain the minimax algorithm, and see how alpha-beta pruning could help in reducing the number of nodes that need to be searched to find the "best" move. For the sake of clarity, all nodes except the leaf nodes have been labeled in this case (A, B1, etc). Given the game search-tree shown in Figure 1.2, on the next page, alpha-beta pruning works as follows:

Look-Ahead Depth	Number of Leaf Nodes
1	8
2	64
3	512
4	4,096
5	32,768
6	262,144
.	.
20	1,152,921,504,606,846,976

Table 1.2: An Example of Exponential Increase in Number of Leaf Nodes to be Searched with Increasing Number of Plys Searched in checkers for Non-Capture Moves

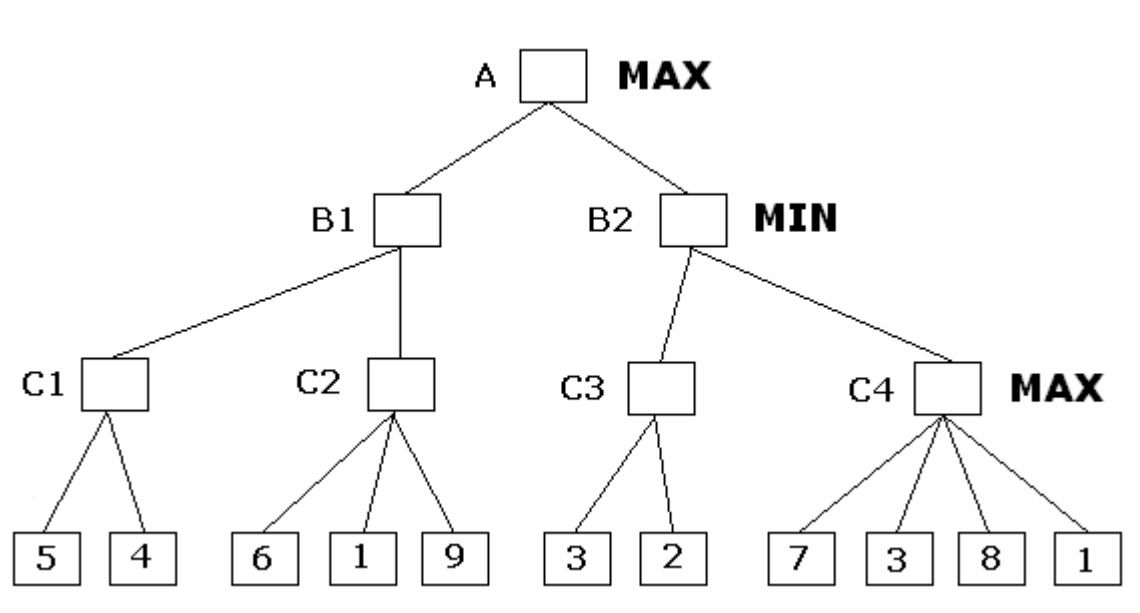

Figure 1.2: Example Game-Tree before Alpha-Beta Pruning

Just as in the case of ordinary minimax, the algorithm begins by examining the scores returned by the position evaluation function for the bottom-leftmost leaf nodes, which in this case, are the children of node C1, which is a max node. Since it is at max-level, the maximum of 5 and 4 - 5 is returned to C1, which is in turn, temporarily stored in B1, which is at min-level. The algorithm examines the children of C2 next, and comes across the value 6. Since C2 is at max-level, we can safely say without examining its remaining children that it would return a value that is greater than or equal to 6. However, B1, which is C2's parent, is at min-level, already has a value of 5 returned from C1 and would choose the lower value, which in this case is 5. Thus, it is safe to ignore the remaining children of C2, leaving the value of 5 intact in B1.

Now, we examine the children of C3, and since it is at a max-level, the greater of 3 and 2 - 3 is returned to C3, which in turn, is temporarily stored in B2. Though the nodes B1 and B2 are at min-level, their parent node A is at max-level, and will always choose the greater

value among B1 and B2. We know that B1 contains 5, and since B2 now temporarily holds 3, even if we examine all the children of B2's other child node C4, the final value in B2 would be a value less than or equal to 3. Since A is a max-node, it would thus choose B1, since its value will now be higher than that of B2, regardless of what values C4's children might have. Thus, the algorithm would ignore the entire branch C4, and choose the move B1, with score 5. This is illustrated in Figure 1.3.

1.2.3.3 Effect of Move Ordering on Performance

It should be noted that the number of leaf positions evaluated is heavily influenced by the ordering of the moves in the game search tree. In other words, alpha-beta pruning works best when the "best" moves are searched first. In the worst case scenario, with an average branching factor of b, and a search depth of d, the number of leaf nodes evaluated is the same as in the case of normal minimax – b^d , but in the case of optimal move ordering (best moves searched first), the number of leaf nodes searched reduces to $\sqrt{b^d}$. This means that the effective branching factor is reduced to its square root, or in other words, the search can be performed twice as deep with the same amount of computation [3]. In the specific example shown in Figure 1.3, using minimax, the number of leaf nodes searched was 11, while alpha-beta pruning enabled us to obtain the same result while searching only 5 leaf nodes - a reduction of approximately 54.55% in the amount of search conducted. Considering that quite often, the difference between a winning machine and a losing machine is in its computational power (with respect to the amount of instructions it can execute per second, or the depth to which it can evaluate a game search-tree), it is obvious that alpha-beta pruning is an invaluable tool in building an efficient search-based game playing machine.

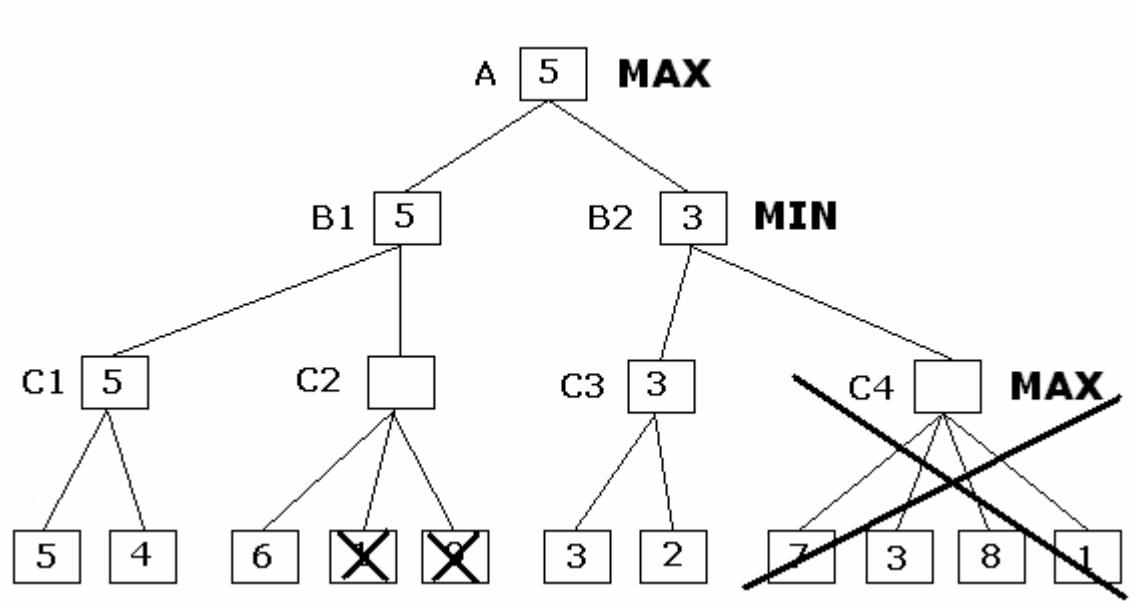

Figure 1.3: Example Game-Tree after Alpha-Beta Pruning

As a side note, "alpha" refers to the minimum value that the MAX player is assured of obtaining, and "beta" refers to the maximum value that the MIN player is assured. Usually, alpha is initialized to negative infinity, while beta, to positive infinity. When at a given node, beta becomes less than alpha, that node and its children are eliminated from the search, since they would not come about, if each player plays perfectly (which is one of the assumptions of the alpha-beta pruning algorithm). Pseudo-code for the algorithm is shown in Algorithm 1.2 on the next page [6].

1.2.4 Optimizing Alpha-Beta Pruning: Iterative Deepening

1.2.4.1 Motivation for Iterative Deepening

As explained in Section 1.2.3 on page 7, Alpha-Beta pruning is a great technique that can yield significant gains when it comes to improving the efficiency of a minimax search of a

game tree. However, its efficacy is highly dependent on move-ordering within the search tree. That is, for optimal results, the best moves should be searched first. Iterative deepening is a simple, yet sometimes counter-intuitive technique that aids in achieving this very result.

Usually, search-based game playing machines have some kind of restriction as to the amount of time available to perform the search before returning the "best" available move. The basic problem here is that we are not sure how deep the search should be. On one extreme, we could specify that the search should be conducted up to the terminal nodes – but this could be too deep, resulting in the allocated time running out before huge swathes of the game tree have even been searched, and returning a potentially sub-optimal move. On the other hand, we could limit the search - depth to an arbitrary value of d plys, but once again, there is no way to say if d is too shallow, or too deep. This is where iterative deepening comes in handy.

```
function minimax(node, depth)
    return alphabeta(node, depth, -∞, +∞)

function alphabeta(node, depth, α, β)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the adversary is to play at node
        foreach child of node
            β := min(β, alphabeta(child, depth-1, α, β))
            if α ≥ β
                return α
```

```
    return β
  else {we are to play at node}
      foreach child of node
          α := max(α, alphabeta(child, depth-1, α, β))
          if α ≥ β
              return β
    return α
```

Algorithm 1.2: Pseudo-Code for Alpha-Beta Pruning

1.2.4.2 Basic Idea

The basic idea is that a depth first search (DFS) is done to depth 1, and if there is still time left, we increase the search depth by one, and so on, until the time allocated to make a decision on the next move (for example, roughly six minutes, in the case of checkers), runs out [7]. Thus, if the branching factor is b, the number of nodes searched would be $b + b^2 + b^3 + \ldots + b^{d-1} + b^d$. Now, it might appear that the earlier searches (for instance, the search to depth 1, depth 2, depth 3, etc.) are wasteful in terms of time. However, the greatest utility of iterative deepening is that the results of each iteration can be used to re-order the nodes attached to the root, with the assumption being that the best moves at depth d, would continue to lead to the best moves at depth d + 1 (This assumption does hold, in most cases). This results in the best moves being searched first with every iteration, thereby approaching optimal results for alpha-beta pruning.

1.2.4.3 An Example

An example of iterative deepening can be seen in Figure 1.4 below. Here, the root node, and its four children A, B, C and D are shown with the corresponding scores returned by the position evaluation function after a search has been conducted to a depth d. Before the next iteration, the child nodes are re-ordered so that the nodes with the higher scores would be searched first. The results of the next iteration (searching up to depth d + 1) can be seen on the right hand side of the figure, which in this case backs up the assumption that the "best" moves at depth d are very likely to help in predicting the best moves at depth d + 1 too. Various techniques are often used in conjunction with iterative deepening in order to increase the efficiency of alpha-beta pruning, and these include using transposition tables, keeping track of best moves from previous searches so that they can be searched preferentially if the same board position is encountered in a future search, using history heuristics [7], etc.

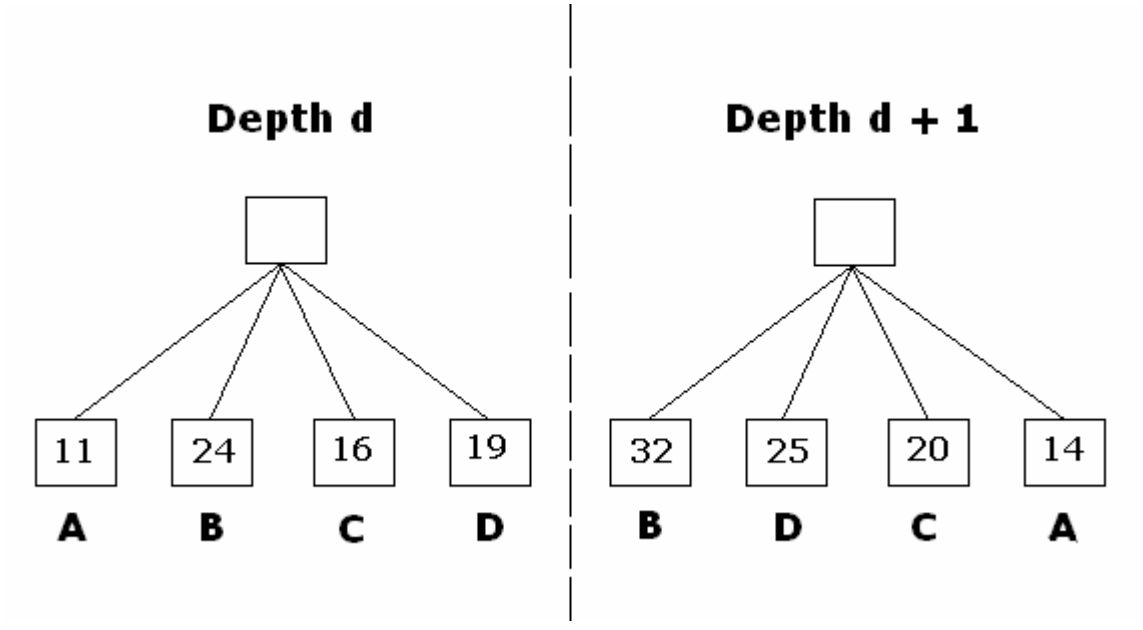

Figure 1.4: Example of Move Re-ordering Using Iterative Deepening

1.2.5 Games and Moves Databases

Another tool that is extensively used by most intelligent game playing systems consists of game and moves databases or books. These come in various assortments and shapes, but mostly fall into three categories:

- Opening Game
- End Game
- Classics

It is common practice to split up or describe chess or checkers games in terms of opening game, middle game and end game. Each stage is characterized by different styles of play that would aid in winning, and the knowledge acquired over the years has led to the development of extensive moves databases, especially in the case of opening and end games.

1.2.5.1 Opening Book

In the case of checkers, a seemingly harmless move at the early stages of a game could result in the game being lost, even with "perfect play" throughout the rest of the match. This is not quite the case in chess, where opportunities for recovery do exist, purely because of the greater freedom of movement that the pieces in a chess game have, with respect to those in a checkers game. Thus, there are literally thousands of so-called "standard" opening move combinations, whose sole purpose is to navigate the tricky waters of the opening game relatively unscathed, or to gain any kind of leverage (material or positional) that could then be used to hurt the opponent during the middle game. Since most players at the expert level spend copious amounts of time and effort memorizing these opening move combinations, it

can be easily seen why a database of such moves would be a great tool for any machine that has aspirations of challenging opponents of that level of expertise.

1.2.5.2 Endgame Databases

Quite similarly, once the game has reached an advanced state and more than likely the number of pieces on the board have been reduced to a handful for either players, the endgame is reached. Unlike other stages of the game (including the opening phase), any error made at this stage will almost certainly result in defeat, especially when playing against an accomplished opponent. Very often, to ensure victory, a player has to manouevre his or her pieces in precise order, ensuring that strategic locations on the board are occupied at the right time, by the right piece so that the opponent has no chance of being let off the hook. Such precise play is rarely accomplished accidentally, and hence the raison d'etre for truly encyclopedic endgame databases.

1.2.5.3 Classic Game Databases

The final class of game or moves databases would be those made up of "classic" games, usually played out by the true masters of the game in question - for example, Bobby Fischer or Garry Kasparov in chess, or Dr. Marion Tinsley in checkers. Such games are usually characterized by seemingly illogical moves (for the casual observer) that lead up to either an astounding assault on the opponent, or staving off defeat from an otherwise lost position. Once again, such moves rarely, if ever, occur to an ordinary player, and hence, someone who has spent time to study them usually has an advantage over an opponent who hasn't, all other things being equal. For these reasons, almost all serious game-playing programs

employ a fairly extensive database of classic games played in the past, to aid in their game-play.

1.3 Chinook

Any discussion on what has been accomplished in the field of artificially intelligent game-playing programs, especially in the case of checkers would be incomplete without mentioning Chinook. A fairly brief overview of what Chinook is and how it was developed follows.

1.3.1 History

1.3.1.1 Origin

In 1952, Arthur Samuel wrote a checkers playing program, that through continuous refinement, actually managed to beat Robert Nealy, a blind checkers master from Connecticut in a famous game in the 1960s. As a result, his program acquired the reputation of being a Masters-level player (though it later turned out that Nealy was not nearly as good at checkers, as claimed), and this in turn, led to almost no work being done in the field of Computer checkers for the next 25 years, since it was apparently, a "solved" problem [1].

In June 1989, Dr. Jonathan Schaeffer at the University of Alberta, Canada, who had been working on computer chess, decided to write a computer program that would be good enough to beat the human checkers champion in the short term, and ultimately, solve the game of checkers - giving the world, Chinook. He worked with a team that included Norman Treolar as the checkers expert, Robert Lake, who was responsible for the endgame databases, and Martin Bryant, who supplied a very large, and very good, opening book. Many graduate students were also contributors to the development of Chinook.

1.3.1.2 Performance in Tournaments

About a year after the project began, in August 1990, Chinook competed in its first two human checkers tournaments and did very well. It won the Mississippi State Open undefeated, and even beat three Grandmasters. While at the US National Open, it placed second, once again defeating three Grandmasters in the process. It should be noted that the US National Open is the biennial event used to determine the next challenger to play for the World checkers Championship, and by coming in second to the reigning World Champion at the time, Dr. Marion Tinsley, Chinook had earned the right to a 40-game match against him for the Championship.

However, this was the first time that a computer program had earned the right to play for a human world championship, and perhaps, for this reason, both the American Checker Federation (ACF) and the English Draughts Association (EDA) did not sanction a Chinook versus Tinsley match. Partly because of their refusal to sanction the match, Dr. Tinsley resigned as World Champion in June 1991. It is worth noting that Dr. Marion Tinsley had been the premier checkers player in the world for the past 40 years, losing just 5 games in that lengthy span, over thousands of games.

Though not officially sanctioned, Chinook played Dr. Tinsley for the "de-facto" world championship in 1992, in London and lost. It won two games, while losing four. They played a rematch in 1994, but after drawing six games, Dr. Tinsley forfeited the match, owing to bad health, and unfortunately, passed away shortly afterwards. In 1995, Chinook took on Don Lafferty, widely acknowledged as the second best human checkers player after Dr. Tinsley, and narrowly defeated him in a very close-fought match with one win and thirty one draws. In 1996, Chinook finished first at the US National Open, and was retired from match-play, since Dr. Schaeffer's team decided that there was nothing left to prove.

1.3.2 Behind the Scenes

Chinook is written in the C programming language, and runs under the UNIX operating system. For the US National Open in 1990, Chinook ran on an IBM RS/6000 Model 530 workstation, with just 32 megabytes of RAM (which was a significant amount in those days) [1]. There are four major components that contribute to Chinook's playing strength:

- Search
- Knowledge
- Endgame database
- Opening book

1.3.2.1 Search

Chinook uses all the standard techniques that were mentioned earlier in this chapter - minimax with alpha-beta pruning, iterative deepening, transposition tables, and a variety of other heuristic techniques. It goes without saying that the deeper a search can be performed within the allowed time frame (usually, 6 minutes), the better the program performs in general. Thus, it becomes important that as much of the available search time be used to investigate promising leads as possible, rather than wasting time on evidently inferior options. The developers of Chinook have spent considerable time and effort in developing suitable heuristics which will aid in extending or truncating a search, as needed. Since in checkers, a material deficit is more significant than in chess, sacrifices of a single piece are common, but ones involving two or more pieces are rare. A technique called forward pruning is used to quickly curtail lines with material deficits. Basically, if a line leads to a material deficit, and there is insufficient positional compensation (that is, the positional

evaluation for that position is below a certain threshold value), then the remaining search depth is reduced to half. If the result of the remaining search does not restore the material deficit, or results in an acceptable positional compensation, then this line is abandoned. Otherwise, the full search depth is restored, and the line is explored completely.

Checkers knowledge is used to extend the search along interesting lines, such as useful captures, checkers running un-opposed to being crowned, some promotion of checkers to kings, and positions where one side is forced to repeat moves (quite common in checkers, unlike in chess). All potential search extensions have to pass through a variety of heuristic tests to ensure that time is not wasted on extensions with little chance of providing useful information. At the US Open, Chinook averaged 20 ply searches (plus extensions). At the start of the game, Chinook searched at least 15 ply; in the endgame, 25 ply searches were common. In contrast, the Deep Thought chess program achieves 10 or 11 ply searches during the middle game using special purpose VLSI hardware. Chinook performs iterative deepening 2 ply at a time to factor out some of the instabilities that arise when searching to odd and even depths. Although 20 ply searches sound impressive, they still fall short of human capabilities, especially during the endgame. Strategic objectives that an expert human opponent can perceive are well beyond the brute-force search horizon, and this is one problem that the Chinook team is working on overcoming.

1.3.2.2 Knowledge

In the case of Chinook, "knowledge" is mostly encoded in terms of its evaluation function. Apparently, at the US Open, the move that was said to be positionally "best" by the program after a 5-ply search (which is the minimum search depth in Chinook) turned out to be the recommended move even after a 20-ply search, implying that the strength of the

program was not based on pure search (although it does make the difference when it comes to competing at an expert level). Additionally, Chinook was able to correctly predict the opponent's move 80% of the time, which would have been ever higher, if the mistakes made by human opponents were factored in. In computer chess, a prediction rate of 50% would be really good, but the fact that the number of pieces and squares in checkers are less makes it easier to predict moves too, as compared to chess.

Chinook divides the game into five phases - opening, middle game, early endgame, late endgame and database. The first four phases each have twenty two user-adjustable parameters that influence the evaluation function [1]. A position evaluation is the linear sum of the 22 heuristics multiplied by their user-defined weights. The last phase has perfect knowledge, and therefore has no parameters. The parameter settings used for the US Open were determined manually. A database of 800 classic Grandmaster games was created and used to develop and automatically tune the program's knowledge. Initial attempts revolved around tuning the parameters in order to maximize the number of times Chinook would select the same move as the Grandmaster, over all games. The very same technique employed by the Deep Thought chess team - which treats the problem as an over-determined system of equations to be solved where one dimension is the number of positions used, and the other is the number of heuristic parameters - was used by the Chinook team, too. It was found, however that more often than not, Chinook selected alternate moves that were either as good as, or even superior to, the Grandmaster moves - which adds to its strength by being able to select moves without biases of human preconceptions, thereby allowing Chinook to continue to surprise the human opponent.

1.3.2.3 Endgame Database

Endgame databases are a computer-generated collection of positions with a proven game-theoretic value, and were pioneered in Computer chess. At the US Open, Chinook had access to the entire 6-piece database, comprising roughly 2.5 x 10^9 positions. What this meant is that whenever a position with 6 pieces or less was encountered, the program could look it up in the database, and return an exact win, loss or draw value. In computer chess, endgame databases are of limited utility since most games never get far enough to utilize them. In checkers, however, since capture moves are forced, some positions near the start of the game can be analyzed right to the end of the game. For instance, a nominal 15-ply search at the start of the game is sufficient for Chinook to start reaching positions in the endgame databases. At the time of writing, the Chinook team had announced that they had succeeded in building the entire 10-piece database. Computing such enormous databases and efficiently representing them in a compact form that can be used in real-time are both challenging issues.

Though the endgame databases represent perfect knowledge, that does not necessarily guarantee perfect play. Chess databases often store distance to win (or mate), for each position. Once in a winning database position, the program can win by only playing those moves from within the database that minimize the number of moves to win. Since the checkers databases are used throughout the game (even at the beginning), they have to be compressed enough to fit into RAM. To enable this, the Chinook team only stores the results of a position, not its distance to win. This means that Chinook could find itself in a winning 6-piece position, but will still have to search to find the winning line. A lot of techniques have been tried to reduce the size of the endgame database, including neural networks to build an evaluation function that can predict the game theoretic value of a

position with a high degree of reliability - which would enable the team to simply store those positions for which the function returns an incorrect prediction in the database.

1.3.2.4 Opening Book

The standard, but scientifically uninteresting way of generating an opening book is to manually type in thousands of positions from human-authored books. The consequences of the lack of an opening book are much more serious in checkers than in chess. In chess, a player using only his experience during the opening phase may play inoptimal moves, and end up in an inferior position. In checkers, however, many opening moves, including as early as move 4 in a game have been analyzed, and found to be forced losses (in other words, from that position, barring a blunder on the opponent's part, even perfect play would result in a loss). Without an opening book, a human or computer player runs the risk of falling into one of these traps, and obtaining a lost position very quickly. Attempts to create a machine-generated opening book yielded less than satisfactory results, and thus, the Chinook team decided to create an "anti-book", a book not of moves to play, but of positions to avoid [1]. Anti-book positions were collected from human opening books, books on opening traps, Grandmaster games, and from Chinook's own games and at the time of the US Open, there were roughly 2000 such positions in Chinook's opening book.

The majority of the information about Chinook's internals is from at least ten years ago. Technology has advanced by leaps and bounds in the meantime, and it is known that they now have the entire 10-piece endgame database at their disposal. It is quite clear that the combination of efficient brute-force search, along with continuous refinement of the various heuristic parameters used in the position evaluation function, and perfect knowledge of the

endgame, along with a formidable opening book make Chinook one of the best, if not the best checkers players in the world.

CHAPTER TWO
SOAR CHECKERS

2.1 Philosophy Behind Soar Checkers

Brief Introduction to Soar - Soar is an architecture designed to aid in developing general intelligent systems [8]. It has been in use for over twenty years and some of its applications have been in building flight simulators, AI engines for interactive 3D games such as Quake, etc. A more detailed description of Soar can be found in Appendix B.

Motivation for Soar Checkers - The superiority of game playing systems based on the minimax algorithm such as Chinook, as explained in Section 1.2 on page 2, is almost entirely due to treating games as a search problem and the increased availability of inexpensive computing power. While this approach is highly effective in building championship calibre AI systems, a sizable group within the AI community share the opinion that a truly "intelligent" system should not rely mainly on raw number-crunching in order to exhibit intelligent behavior. Instead of replicating the techniques used by existing AI game playing systems, it was decided to explore the idea of building a system that relies almost exclusively on tactics and strategy for game play.

Basic Philosophy - The basic philosophy behind Soar Checkers is that it should be possible to create a game playing system that would beat novices with ease and at least challenge advanced novice to intermediate-level players by designing a rules-based expert system whose knowledge base consists of nothing more than the rules of checkers, and a set of guidelines for game-play based on good strategy. In this chapter, the user interface designed for Soar Checkers is described first, in Section 2.2 and then, an explanation of how

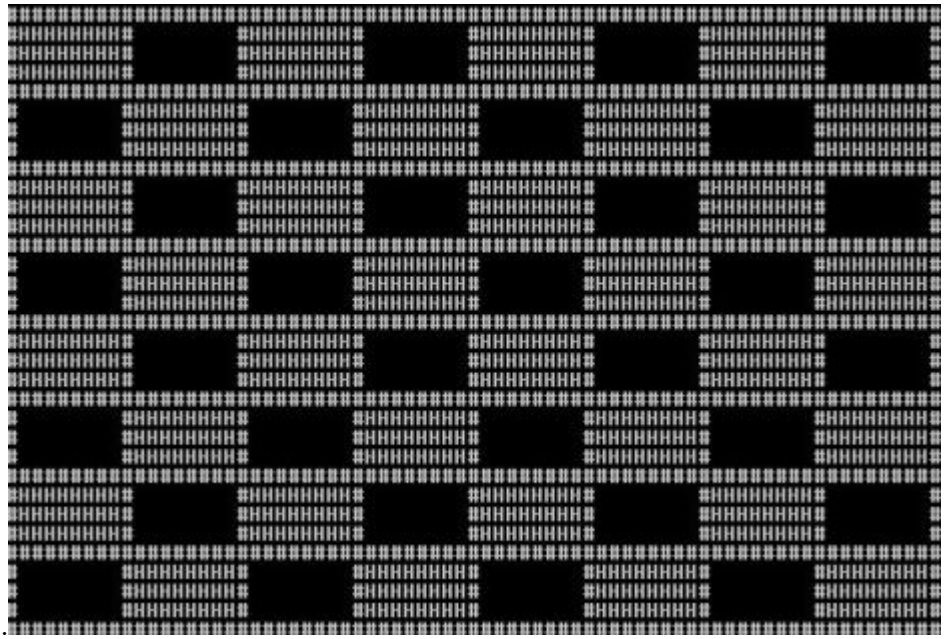

Figure 2.1: An Empty Checker Board in Soar Checkers

Soar Checkers works follows in Section 2.3. The strategies used to build Soar Checkers are explained in fairly good depth on page 60

2.2 The User Interface

2.2.1 The Game Board

For its simplicity, ASCII art has been used to represent the checkers game board. Though the board is of size 8 x 8, only half of those squares are valid in checkers. Thus, valid squares are either blank, or have a piece in them, while invalid squares are filled with a 'H'. There are only four different kinds of pieces[4] in checkers, and they are represented as:

- Red Checker – 'RC'

[4] There is some confusion prevailing as to whether the pieces are colored black and white, or red and white. Please refer to Appendix A for more details.

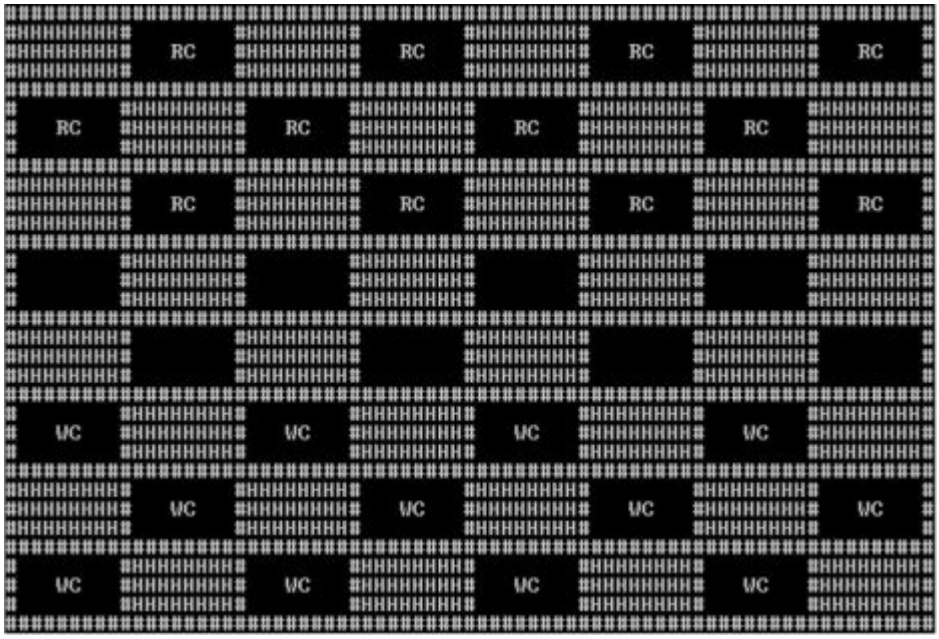

Figure 2.2: Initial State of the Game Board in Soar Checkers

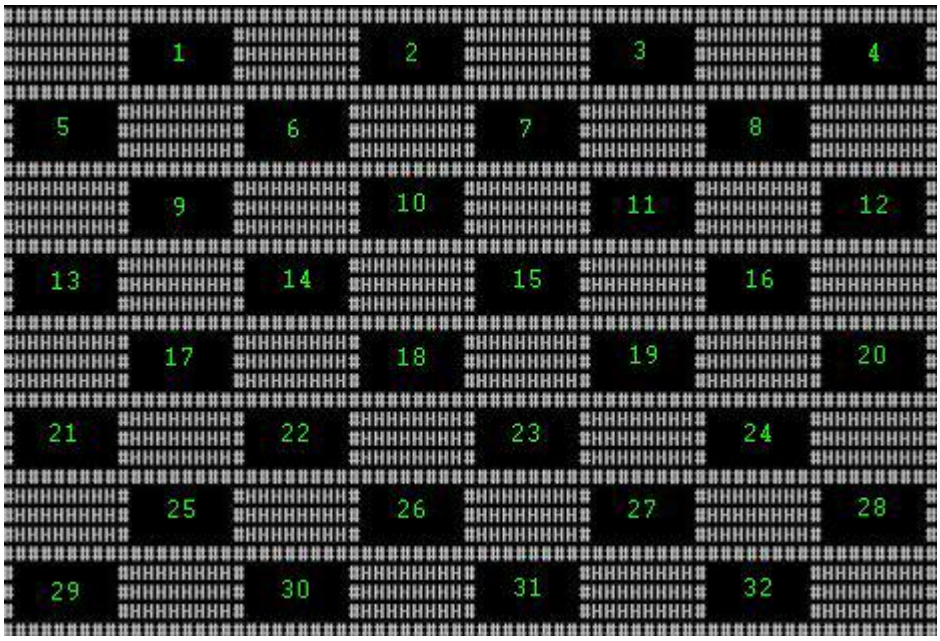

Figure 2.3: The Numbering Scheme Used for the Board Squares in Soar Checkers

- Red King – 'RK'
- White Checker – 'WC'
- White King – 'WK'

Figure 2.1 shows what the game board looks like, devoid of any pieces. As explained above, a piece can be legally moved only to the dark squares. Figure 2.2 shows the state of the game board, before any moves have been made, assuming that red moves from top to bottom. Figure 2.3 shows the numbering scheme used in Soar Checkers, which is the same as the classic numbering scheme used in checkers literature to describe various board positions and moves.

Modes of Operation - On starting up, Soar Checkers presents the user with three modes of operation:

1 Soar agents versus Soar agent

2 Soar agent versus Human

3 Human versus Human

Choices one and two are the ones of particular interest. In the first case, two Soar agents are created - Red and White, corresponding to the color of their pieces. As per the rules of checkers, the Red agent moves first. After each agent move, the new board position is displayed, and the game is paused until the user presses any key. Without this feature, it would be almost impossible to keep track of each agent's moves since an agent usually needs only a few seconds to make a move.

If the user opts for the third mode of operation, the board is automatically initialized with the Red piece at the top, and the White pieces at the bottom, as shown on Figure 2.2. The players proceed to move or jump as described in Sections 2.2.2, 2.2.3, 2.2.4 and 2.2.5.

If the user chooses the second mode of operation, he or she would be presented with two more choices:

- The color of his or her pieces
- If he or she wants to move from top to bottom, or bottom to top

Once these choices are made, the board is initialized accordingly, and whoever is red (human or agent) moves first. When it is the human players' turn, the following line is displayed, assuming that the human is playing as red:

"You are Red .... Enter the number of jumps / moves (1 for single move/jump, 2 for double jump etc...)"

There are four possible kinds of moves, and each of them would be explained in detail in the subsequent sections.

2.2.2 Making a Move

To make a normal move (that is, not a jump), the user enters "1" at the previous prompt. This results in the following prompt being displayed:

"Enter source (1 - 32): "

As already explained, the valid squares on the game board are numbered from 1 to 32 using the numbering scheme shown in Figure 2.3 on page 28. Thus, if the user wants to

move his piece located at square 12, he would enter 12 here. Now, a new prompt is displayed:

"Enter middle (-1 if you're just making a move, 1 - 32 if it's a jump): "

Since this is a normal move, the user should enter -1. The next prompt would be:

"Enter destination (1 - 32): "

If the desired destination of the move is square 16, all the user has to do is enter 16. If the user decides to quit the game, he or she could enter "quit" at any of these three prompts.

An Example - Suppose the game is in the initial state, as shown in Figure 2.2 on page 28. The user's color is red, and to move from square 12 to square 16, he would have to enter:

"Enter source (1 - 32) : 12"

"Enter middle (-1 if you're just making a move, 1 - 32 if it's a jump) : -1"

"Enter destination (1 - 32) : 16"

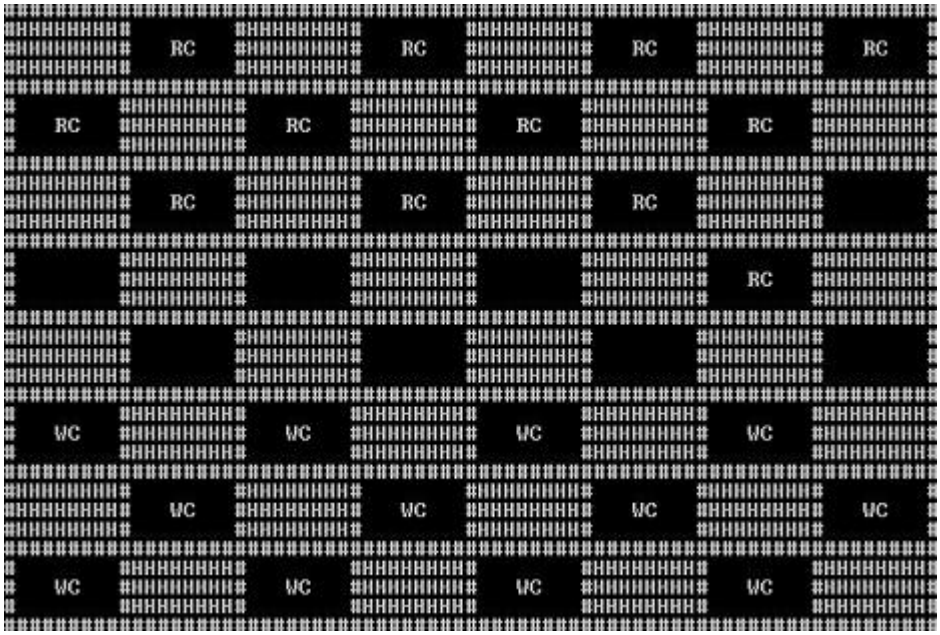

Figure 2.4: Example for Making a Normal Move: Board Position After Moving from Square 12 to Square 16

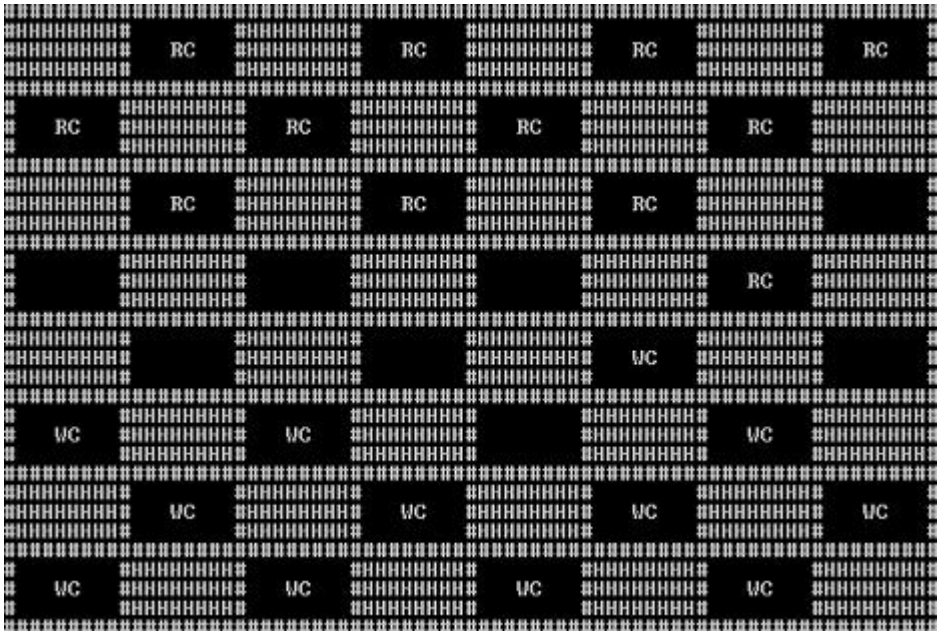

Figure 2.5: Example Board Position Before a Single Jump

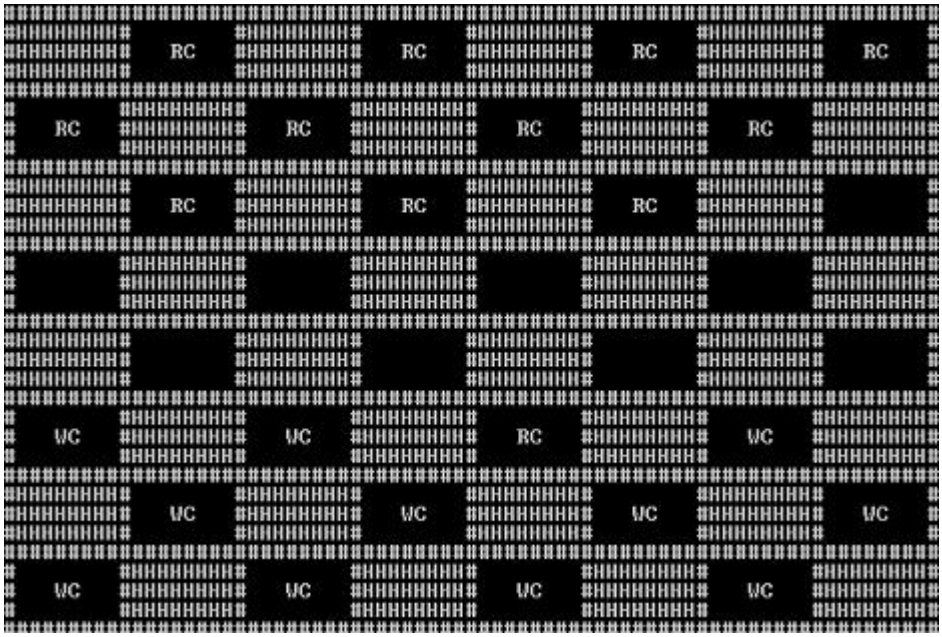

Figure 2.6: Example Board Position After a Single Jump

2.2.3 Making a Single Jump

This is very similar to making a normal move, as explained in Section 2.2.2. on page 30. The only difference is that at the second prompt:

"Enter middle (-1 if you're just making a move, 1 - 32 if it's a jump): "

Instead of entering "-1", as is done in the case of a normal move, the number of the square which contains the opponent's piece that the user desires to jump over, should be entered. This will be illustrated in the following example.

An Example - Figure 2.5 shows the board position when it is the Red players' turn to make a move. If the player desires to make a single jump over the white checker located at square 19, the source is 16, the middle is 19, and the destination is 23. These are the commands that the user would have to enter in order to accomplish that:

"Enter source (1 - 32): 16"

"Enter middle (-1 if you're just making a move, 1 - 32 if it's a jump): 19"

"Enter destination (1 - 32): 23"

The resulting board position is shown in Figure 2.6 on the previous page.

2.2.4 Making a Double Jump

When the user desires to make a double jump, as opposed to three squares in the case of a single jump, five squares are involved - one source square, two middle squares corresponding to the locations of the two opponent pieces that would get jumped, and two destination squares corresponding to the first and second jumps. For both a normal move and a single jump, the user enters "1" at the prompt:

"You are Red .... Enter the number of jumps / moves (1 for single move/jump, 2 for double jump etc...)"

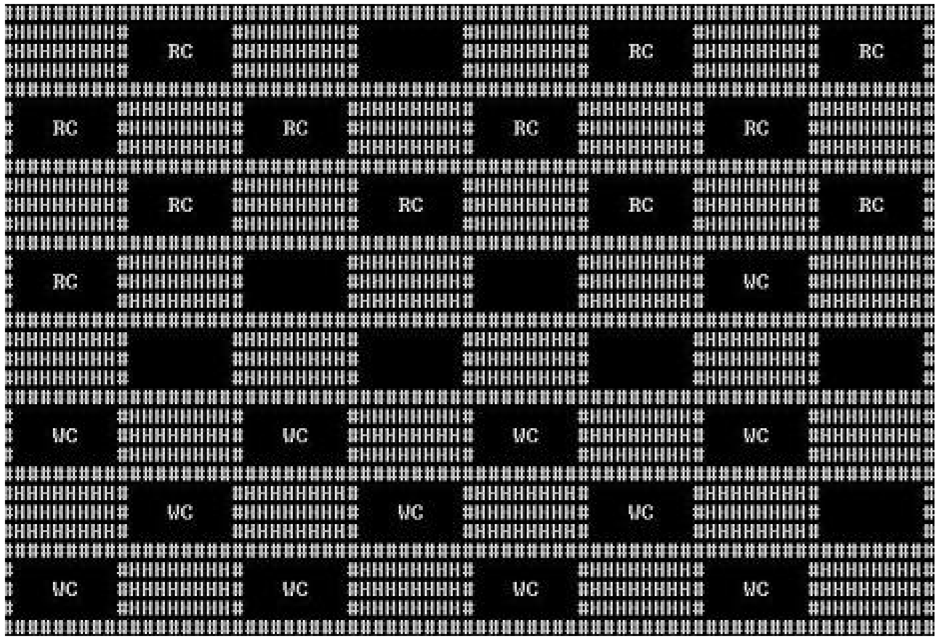

Figure 2.7: Example Board Position Before a Double Jump

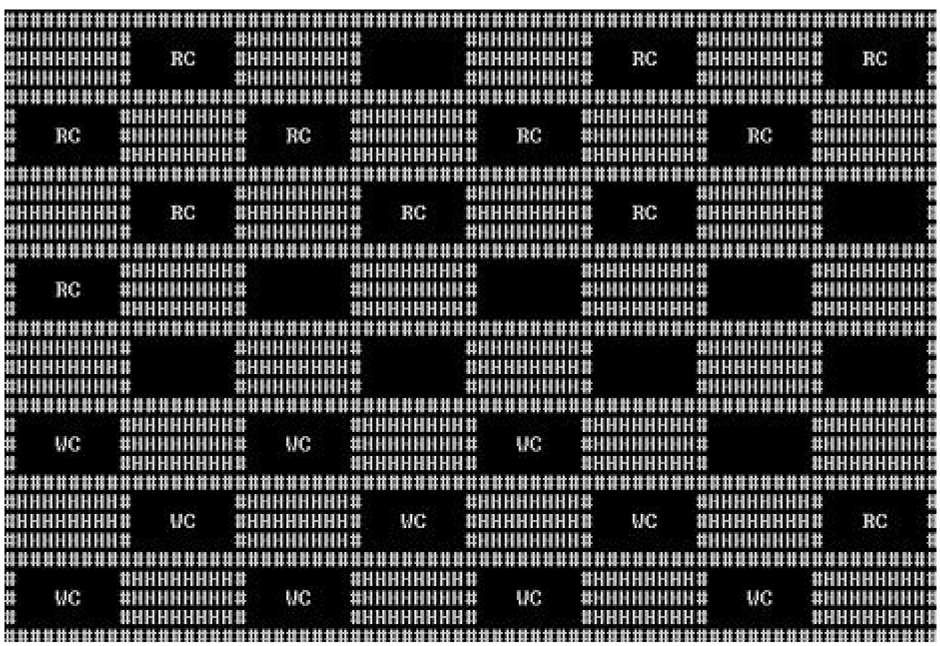

Figure 2.8: Example Board Position After a Double Jump

An Example - However, in order to make a double jump, a "2" should be entered. Let us consider an example of a Red player trying to make a double jump. Figure 2.7 shows the board position when it is the Red players' turn to move. If the user desires to jump the opponents' pieces located at squares 16 and 24, the corresponding source would be 12, first middle would be 16, first destination would be 19, second middle would be 24, and final destination would be 28. These are the commands necessary for the user to do that:

"You are Red .... Enter the number of jumps / moves (1 for single move/jump, 2 for double jump etc...) 2"

"Enter source (1 - 32): 12"

"Enter 1st middle (-1 if you're just making a move, 1 - 32 if it's a jump): 16"

"Enter 1st destination (1 - 32): 19""

"Enter 2nd middle (-1 if you're just making a move, 1 - 32 if it's a jump): 24"

"Enter final destination (1 - 32): 28"

Figure 2.8 on the previous page shows the resulting board position.

2.2.5 Making a Triple Jump

In the case of a triple jump, seven squares on the game board are involved, as opposed to just five in the case of a double jump, and three in the case of a normal move or a single jump - one source square, three middle squares corresponding to the locations of the three opponent pieces that would get jumped, and three destination squares corresponding to the first, second and third jumps. In order to make a triple jump, the user should enter "3" at the following prompt:

"You are Red .... Enter the number of jumps / moves (1 for single move/jump, 2 for double jump etc...)"

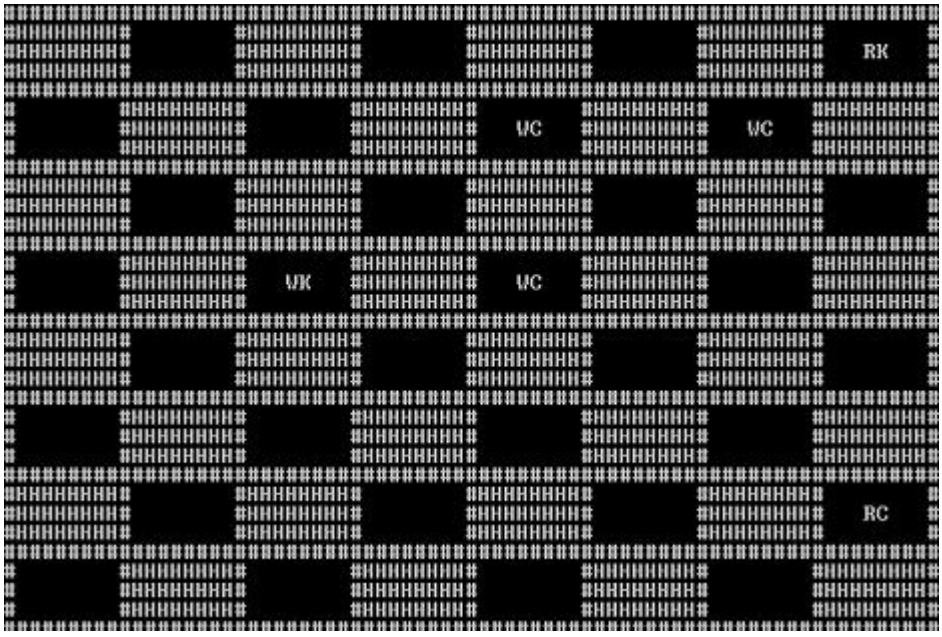

Figure 2.9: Example Board Position Before a Triple Jump

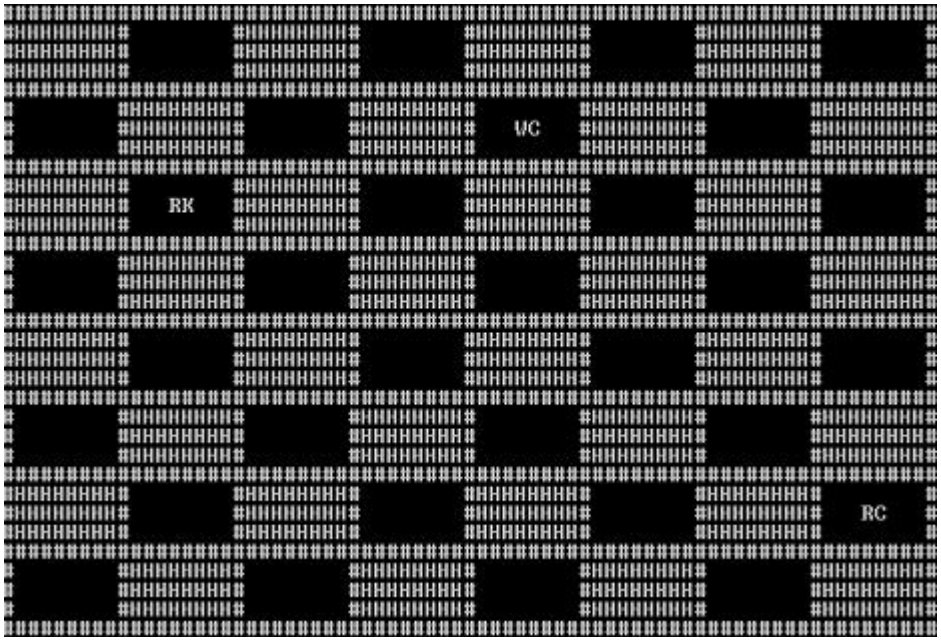

Figure 2.10: Example Board Position After a Triple Jump

An Example - Let's consider an example of a Red player trying to make a triple jump. Figure 2.9 shows the board position when it is the Red player's turn. Unlike a checker, a king can move both forwards and backwards. Thus, the red player could use the red king at square 4 to jump over the white checkers at squares 8 and 15, and then jump backwards over the white king at square 14. In this case, the source would be 4, first middle would be 8, first destination would be 11, second middle would be 15, second destination would be 18, third middle would be 14 and the final destination would be 9. These are the commands necessary for the user to do that:

"You are Red .... Enter the number of jumps / moves (1 for single move/jump, 2 for double jump etc...) 3"

"Enter source (1 - 32): 4"

"Enter 1st middle (-1 if you're just making a move, 1 - 32 if it's a jump): 8"

"Enter 1st destination (1 - 32): 11"

"Enter 2nd middle (-1 if you're just making a move, 1 - 32 if it's a jump): 15"

"Enter 2nd destination (1 - 32): 18"

"Enter 3rd middle (-1 if you're just making a move, 1 - 32 if it's a jump): 14"

"Enter final destination (1 - 32): 9"

The resulting board position is shown in Figure 2.10 on the previous page. In the following sections of this chapter, a detailed description of how Soar Checkers is built under the hood ensues.

2.3 Soar Checkers Internals

Soar Checkers is an expert system with an agent-based architecture built using Soar 8.6.2 and the C++ programming language. The Soar Markup Language (SML) [9, 10] has been

used to interface between the Soar kernel and the game environment written in C++. The real "brains" of the system are encoded as a set of rules (known as productions in Soar parlance), for agents in Soar. The Soar kernel can be thought of as an environment that hosts the Soar agents and provides the various functionalities essential for proper operation of the agents.

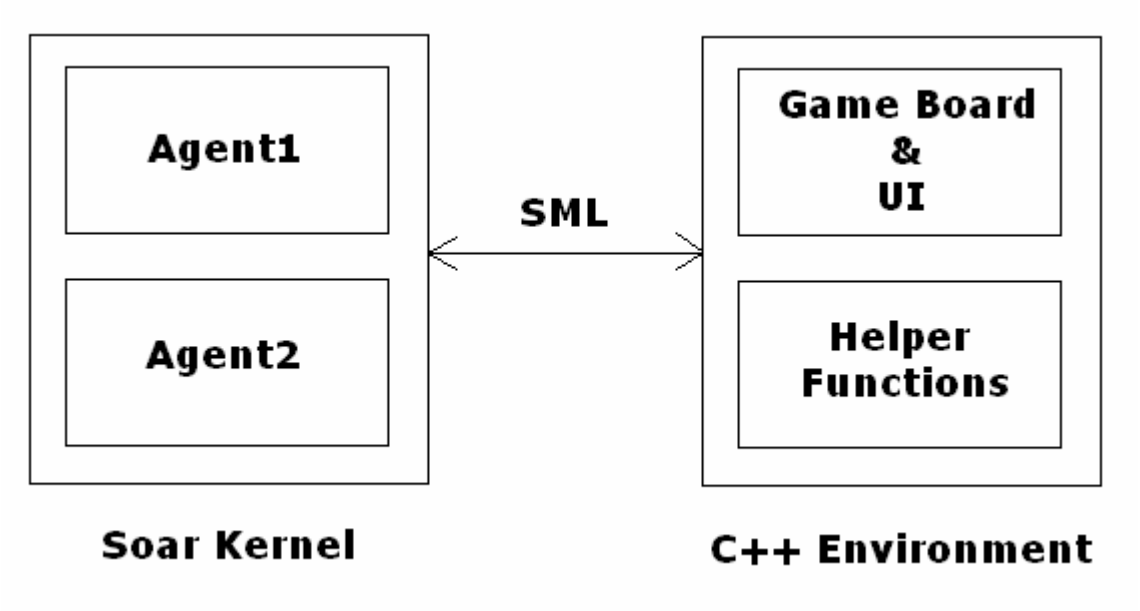

Figure 2.11: Overview of Soar Checkers Architecture

The Game Environment - The game environment, written in C++, accepts user input, interfaces with the Soar agent(s) (in other words, acts a third-party that enables multiple agents in the Soar kernel to communicate), keeps track of, and updates the game state. In addition, some helper functions have also been written in C++ (for example, path-finding algorithm to determine promising attacking options towards the endgame) that provide suggestions to the agents with respect to "good" move options. The agents use their

respective ^io.input-link and ^io.output-links[5] to communicate with the game environment. An overview of the architecture used to build Soar Checkers can be seen in Figure 2.11 on the previous page. The various steps involved in the operation of Soar Checkers are explained in subsequent sections.

2.3.1 Initialization

2.3.1.1 Soar versus Soar

The game board, which is part of the game environment, is initialized as shown in Figure 2.2 on page 28. An instance of the Soar kernel is created in a new thread, and two Soar agents (with potentially different sets of rules) "Red" and "White" are created. Since red always goes first, one of the attributes on the ^io.input-link called "isTurn" is set to 1 for the Red agent, while the corresponding attribute for the White agent is set to 0. Certain data structures are created in the agents' working memories using Soar elaboration rules. These data structures encode information about how the various legal squares on the game board are connected to each other, the location of the king rows, the kinds of pieces in the game etc.

2.3.1.2 Soar versus Human

In this case, an instance of the Soar kernel is created in a new thread, and a Soar agent is created (named "Red" if the human chose to be the white player, or "White" if the human chose to be red). Depending on the agent's color, the "isTurn" attribute on its ^io.input-link is set to 1 or 0 (1 if it's Red and 0 otherwise). The remaining initialization steps in Soar are very similar to those described in Subsection 2.3.1.1. Meanwhile, an instance of a class

[5] For more information on these, please refer to Appendix B

named “Human” is created in the game environment in C++ and is used to accept the human player's moves and make corresponding changes in the board position, among other things.

2.3.2 Generate all Legal Moves

The basic goal of an agent in Soar Checkers is to make moves in a given game that would lead to a win or a draw. In order to do that, the first thing an agent does on its turn to move, is to generate a list of all possible legal moves and jumps, given the current board position - which is sent to the agent on its ^io.input-link by the game environment. The list of moves thus generated might have four different kinds of moves:

- A normal move
- Single jump
- Double jump
- Triple jump

Each move generated at this stage are added to the top state of the agent so that they are easily accessible at later stages of the decision making process. Since these moves are generated as a result of an operator application, they have o-support, and remain in working memory until explicitly removed [8]. A move generated at this stage is represented thus:

```
(<s>          ^legalMoves     <aMove>
              ^mobileCells    <pos> )
(<aMove>            ^numJumps 1
              ^source         <pos>
```

```
^midSquare      -1
^destination    <movepos>
^piece          <checker>
^savingSquare       -1
^riskBeforeMove     unKnown
^riskAfterMove      unknown )
```

The above segment of code adds a data structure called "legalMoves" to the top state in working memory (represented as <s>). Within this data structure are a series of attribute-value pairs which hold information specific to each move. The ^source, ^midSquare and ^destination attributes correspond directly to locations on the game board (midSquare will be -1 in the case of a normal move). ^numJumps is 1 for both a normal move and a single jump, 2 for a double jump and 3 for a triple jump. ^piece refers to the kind of piece involved in the move. ^savingSquare holds the location of the square on the game board that could potentially save a piece under threat (-1 if none exists). ^riskBeforeMove and ^riskAfterMove are attributes used to identify risks associated with a move, and will be explained in more detail in subsequent sections.

An Example – Let us consider an example. Given the board position shown in Figure 2.9 on page 36, and assuming that it is the Red agent's turn, the list of all possible moves and jumps would consist of:

- Single Jump - 4 - 8 – 11
- Double Jump - 4 - 8 - 11 - 15 – 18
- Triple Jump - 4 - 8 - 11 - 15 - 18 - 14 – 9

- Move - 28 - -1 – 32

As a side note, considerable difficulty was faced while writing the rule to generate triple jumps for a king. For reasons that are yet to be understood by the author, the rule for the king, which was quite similar to the one written for triple jumps for checkers except for the fact that kings can jump in both directions, caused the entire program to hang, and finally crash as the system ran out of virtual memory (over 2 GB) - which might very well suggest that an infinite recursion could be the underlying cause, though all attempts to debug the offending rule met with failure. What was really perplexing was that a very similar rule for generating double jumps for kings posed no problems at all. The problem was finally overcome by splitting the one rule into eight separate rules corresponding to the eight different ways in which a king could triple jump:

- Forward - Forward – Forward
- Forward - Forward – Backward
- Forward - Backward – Forward
- Forward - Backward – Backward
- Backward - Forward – Forward
- Backward - Forward – Backward
- Backward - Backward – Forward
- Backward - Backward – Backward

2.3.3 Evaluate Risks Before Move

The next step in the decision making process is to evaluate the risks faced, if any, for each piece involved in the moves that were generated, before actually making the move. In addition, the risks faced by those pieces which cannot move (henceforth referred to as immobile pieces) are also evaluated. At this stage, only the most obvious risks are considered. For instance, in the case of a mobile piece, the following situations put it under risk:

- One of the opponent's pieces is in a square directly in front of it, and the square directly behind it in the same diagonal is empty.
- An opponent king is in a square directly behind it, and the square directly in front of it in the same diagonal is empty.

2.3.3.1 An Example

An immobile piece is under risk when the situation described in the first of the two scenarios above exists. Consider the board position shown in Figure 2.9. If it is the red player's turn to move, none of his pieces are under risk. However, if it is the white player's turn to move, the piece located at square 8 is under threat by the red king at square 4. There is only one possible move for the white player that involves the piece at square 8 -> 8 - -1 - 3. This information will be used to modify the "^legalMoves" entry corresponding to that move generated at the previous stage of the decision making process to reflect the fact that there is a risk before making that move. The "^legalMoves" data structure has an attribute named ^riskBeforeMove, which is set to "unKnown" by default. When a risk is discovered for a move at this stage, the corresponding ^legalMoves is copied to a new data structure

named "^beforeEvaluatedMoves", its ^riskBeforeMove attribute is set to "yes" and the ^savingSquare attribute is changed to reflect the value of the square that could potentially prevent it from being jumped. The old ^legalMoves entry is then deleted. The relevant portion of the resulting ^beforeEvaluatedMoves structure for the move from square 8 to square 3 would then be:

```
^source 8
^midSquare -1
^destination 3
^savingSquare 11
^riskBeforeMove yes
^riskAfterMove unknown
```

2.3.3.2 Homogenizing

Once all the ^legalMoves have been checked to see if any of the pieces involved have a risk before making that move, there would be a set of the original ^legalMoves, and possibly, a new set of ^beforeEvaluatedMoves (if any of the ^legalMoves had a risk before moving). In the next step, an operator[6] called "convertLegalMovesToBeforeEvaluatedMoves" is proposed to copy all the remaining ^legalMoves to ^beforeEvaluatedMoves, and then to delete the ^legalMoves. At the end of this stage, there would remain only a set of ^beforeEvaluatedMoves, some with their ^riskBeforeMove attribute set to "yes", while others would still have it set as "unknown".

[6] For more information on operators in Soar, please refer to Appendix B.

2.3.4 Evaluate Risks After Move

At this stage of the decision making process, for each move generated, a one step look-ahead is performed, its assumed that the move is made, and then the risks faced, if any, for each piece involved in those moves are evaluated - without actually making the move. Unlike in the previous stage, where the resulting data structure ^beforeEvaluatedMoves differed from its corresponding ^legalMoves data structure (which was originally created) only in that its ^riskBeforeMove attribute might have changed to a value of "yes", instead of the default value of "unKnown", in this stage, there are a variety of values that could potentially be assigned to the ^riskAfterMove attribute. Depending on the results of evaluating risks, these are the possible values for the ^riskAfterMove attribute:

- "yes" - Definite risk of being at least single – jumped
- "potentialSingleJump" - Move could potentially result in a single jump
- "singleJump" - Risk of being single jumped
- "doubleJump" - Risk of being double jumped
- "unKnown" - When no known risk is found

2.3.4.1 Scenarios for a Move Being Risky

The number of scenarios corresponding to each risk listed above are too numerous to cover completely here, but lets consider an example each for each risk type. Let ^source be the source square for the move, and ^destination be the destination square for the move. In that case,

- If ^source is one of the corner squares, and there is an opponent piece in the square directly in front of the destination, in the same diagonal as the source, making this move would result in getting at least single jumped. Thus, the ^riskAfterMove attribute would be set to "yes".
- Call the square directly in front of ^destination "^frontDestination". If ^frontDestination is empty, and it has two squares directly in front of it, both of which contain opponent pieces, then making this move would allow the opponent to attack the piece at ^destination by moving the opponent piece that is currently in the square that is not in the same diagonal as ^destination and ^frontDestination, to ^frontDestination. While this is not a certain threat (for example, the threat could be defused by moving the piece from ^destination to the square directly in front of ^destination that is not ^frontDestination provided it is empty), it could potentially result in being single jumped, and hence, the ^riskAfterMove attribute for such a move is set to ^potentialSingleJump.
- The conditions which result in the ^riskAfterMove attribute being set to "singleJump" are very similar to the case described above, except that ^source could be any square (not restricted to just corner squares), the square directly in front of ^destination in the same diagonal as ^source has an opponent checker[7] in it, and the square(s) directly behind ^source is, or are:

    - Empty
    - Have opponent piece(s)

[7] The fact that a king can move in two directions makes the conditions to be tested in an evaluation much more complicated than in the case of a checker.

    - Corner square(s)
    - Has the player's own piece(s), but the square(s) directly behind those piece(s) in the same diagonal as ^source is, or are not empty (thus, preventing a double jump).

- If there is an opponent piece in the square directly in front of ^destination, in the same diagonal as ^source, and there is at least one square directly behind ^source (call it ^behindSource) which has one of the player's pieces in it, and the square directly behind it is empty, and lies in the same diagonal as ^behindSource and ^source, then making this move would result in getting double jumped. The ^riskAfterMove attribute would therefore be set to ^doubleJump.

It should be noted that there is no separate case for a move that would result in getting triple jumped. This is because the conditions for a triple jump to occur, though relatively rare, are a superset of those for a double jump, and detecting that making a certain move would result in getting double jumped, gives us enough information to try and avoid making that move, thereby eliminating the need to write separate rules for triple jumps.

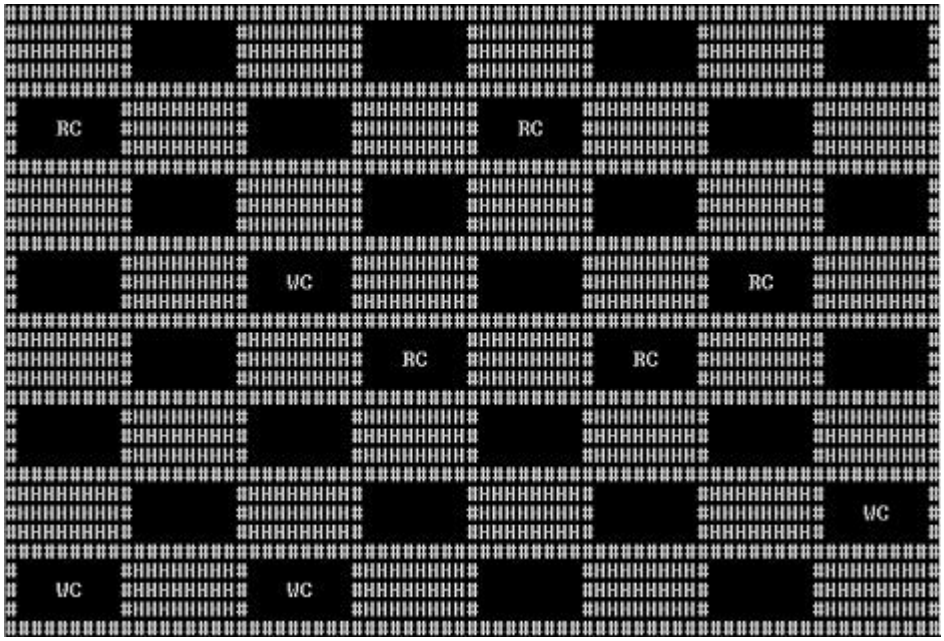

Figure 2.12: Example Board Position Where Some of the Available Move Choices for Red are Risky

2.3.4.2 An Example

Consider the board position shown in Figure 2.12. Assuming that it is the Red player's turn, let us compute the results of evaluating risks after move. There are five red checkers on the board, and at the beginning of this stage, there will be the following ^beforeEvaluatedMoves data structures (only the relevant portions are shown here, for the sake of brevity and clarity), all with their ^riskBeforeMove and ^riskAfterMove set to "unKnown":

1 ^source - 5, ^destination - 9

2 ^source - 7, ^destination - 10

3 ^source - 7, ^destination - 11

4 ^source - 16, ^destination - 20

5 ^source - 18, ^destination - 22

6 ^source - 18, ^destination - 23

7 ^source - 19, ^destination - 23

8 ^source - 19, ^destination – 24

2.3.4.3 Risk Assessment

The 1st move in the list, is from square 5, which is a corner square, and since there is an opponent piece (in this case, a white checker) in square 14 which is directly in front of the destination, and squares 5, 9 and 14 lie in the same diagonal, making this move would result in at least a single jump. Thus, the ^riskAfterMove for this move would be set to "`yes".

The 2nd move would result in getting single jumped by the white checker at square 14, but no worse (double or triple jump) since both of the squares behind the source square are empty. So, ^riskAfterMove for this move would be set to "singleJump". The 3rd move does not result in any apparent loss, and hence, its ^riskAfterMove would remain "unKnown". Similarly, the 4th move's ^riskAfterMove is not changed from "unKnown".

In the case of the 5th move, the destination is 22, and one of the two squares in front of it, specifically square 25 has both squares in front of it (squares 29 and 30) occupied by opponent pieces. Thus, if this move is made, the opponent could attack the red checker that would now be at square 22 by moving the white checker at square 30 to square 25. Of course, in this specific case, this would not result in a loss for Red since square 26 is empty, and the red checker can be safely moved there to avert the threat, but that need not be the case always. Hence, the ^riskAfterMove for this move would be set to "potentialSingleJump".

The 6th move does not result in any immediate losses, and so, its ^riskAfterMove is "unKnown". Similarly, the 7th move has a ^riskAfterMove of "unKnown" too.

If the 8th move is made, square 28 which is directly in front of ^destination and is in the same diagonal as ^source, has an opponent piece in it, and square 16, which is directly behind source has a Red checker in it while square 12, which is directly behind square 16 and is in the same diagonal as ^source, is empty - a double jump would result with ^source 28 and ^destination 12. The corresponding ^riskAfterMove is thus set to "doubleJump".

2.3.4.4. Homogenizing

Once the risks after making a move have been evaluated, if it is found to be anything other than "unKnown", the contents of the corresponding ^beforeEvaluatedMoves data structure are copied to a new data structure that would be added to the top-state, called ^afterEvaluatedMoves, its ^riskAfterMove attribute is changed to reflect the newly computed value, and then the existing ^beforeEvaluatedMoves data structure is deleted. Quite similar to the previous stage, at the end of this stage, an operator called "convertBeforeEvaluatedMovesToAfterEvaluatedMoves" is proposed to copy any remaining ^beforeEvaluatedMoves with their ^riskAfterMove set to "unKnown" to corresponding ^afterEvaluatedMoves data structures with "unKnown" as the value for their ^riskAfterMove, and then delete the original ^beforeEvaluatedMoves data structures.

2.3.5 Saving Helpless Pieces

At the beginning of this stage, the top-state contains a set of ^afterEvaluatedMoves data structures, which contain information about various moves, their risks before making the move, and risks after making the move. However, they do not take into account the effect of

certain class of moves on potentially saving an otherwise "helpless" piece, or the fact that while moving a certain piece may pose no risk to that specific piece, it might result in an otherwise protected piece being jumped. Such issues are explored at this stage, and the agent's working memory updated accordingly.

2.3.5.1 Saving an Immobile Piece

In Section 2.3.3 on page 43 it was briefly mentioned that in addition to pieces that are involved in legal moves, the obvious risks for pieces which do not have legal moves (immobilePieces) are also evaluated. If such a piece exists, a data structure called ^immobilePieceUnderThreat is created, and added to the top-state of Soar. One of the entries in this data structure is an attribute called "^savingSquare", which contains the location of the square which could be used to potentially nullify the threat faced by this piece. If a move (represented as an ^afterEvaluatedMoves data structure) exists such that its destination is the *savingSquare of an ^immobilePieceUnderThreat, then making that move would more likely than not, save that piece. At the same time, we should ensure that the piece involved in that move is not endangered as a result of that move. This condition is easily met when that move's ^afterEvaluatedMoves data structure has its ^riskAfterMove set as "unKnown". Thus, if all these conditions are satisfied, the original ^afterEvaluatedMoves data structure is copied to a new one, its ^riskBeforeMove is set to "yes", and the original structure is deleted. The effect of this change to the contents of the agent's working memory would be explained in Section 2.3.8 on page 60.

2.3.5.2 Detecting Unsafe Moves that Seem to be Safe

Another class of scenarios considered at this stage involves moves which seem to be safe (they have a ^riskAfterMove value of "unKnown"), but actually cause the loss of those, or other piece(s).The number of different cases are too numerous to go over here, but let us consider a few example cases.

2.3.5.2.1 Case One

If there are two pieces A and B such that A is directly behind B, and the square directly in front of B in the same diagonal as A contains an opponent piece (which cannot be jumped by B), a seemingly "safe" move involving piece A would result in at least B getting jumped. Thus, the ^afterEvaluatedMoves data structure corresponding to such a move should not have a ^riskAfterMove value of "unKnown". If the conditions described here are met, the original ^afterEvaluatedMoves data structure is copied to a new one, its ^riskAfterMove is set to "yes" (since we know that making this move would result in at least a single jump though not for this specific piece), and the original ^afterEvaluatedMoves structure is deleted.

2.3.5.2.2 Case Two

If there is an ^immobilePieceUnderThreat data structure with its ^savingSquare at square "A" on the game board, and there exists an ^afterEvaluatedMoves data structure such that its source is square "B", its destination is square "C", "A" happens to be the square directly in front of "C", and A, B and C are in the same diagonal, then making the move from B to C would result in a double jump involving the immobile piece and the piece now at C. Thus, if all these conditions are met, the original ^afterEvaluatedMoves data structure

is copied to a new one, its ^riskAfterMove is set to "doubleJump", and the original structure is deleted.

2.3.5.2.3 Case Three

Another scenario that is considered at this stage is quite similar to the previous case, with a couple of differences - there is no immobile piece involved, and the opponent piece has to be a king. If there is a move with a ^riskBeforeMove "yes", the piece involved in that move is at square "A", there is an opponent king in the square directly behind A, the square D directly in front of A in the same diagonal as the opponent king is empty (which causes the piece at A to have a ^riskBeforeMove of "yes"), and there exists another move with its ^source B, ^destination C , C and D are immediate neighbours (D is either directly in front of, or directly behind C), and B, C and D lie in the same diagonal, then moving the piece from B to C would result in a double jump involving the piece at square A, and the piece now at square C. Thus, if these conditions are met, the original ^afterEvaluatedMoves data structure is copied to a new one, its ^riskAfterMove is set to "doubleJump", and the original one is deleted.

2.3.5.2.4 Case Four

Finally, a potentially devastating situation is checked, where an opponent king could wreak havoc. If there are five squares A, B, C, D and E which are immediate neighbours to each other (B is directly in front of A, C is directly in front of B etc.) and lie along the same diagonal such that squares C and E are empty, square D contains one of the player's pieces, and there is an ^afterEvaluatedMoves data structure with its ^source A and ^destination B, then making that move could potentially result in the guaranteed loss of at least one piece

(either the one in square D, or the one now moved to square B), because this configuration allows an opponent king to be moved to the empty square C, sandwiched between the pieces at squares B and D, placing both of them under threat simultaneously. Thus, if these conditions exist, the original ^afterEvaluatedMoves structure is copied to a new one, its ^riskAfterMove is set to "doubleJump", and the original structure is deleted.

2.3.5.3 An Example

2.3.5.3.1 Initial Structures

Consider the board position shown in Figure 2.13 below. Assuming that it is the Red player's turn, the relevant portions of the ^afterEvaluatedMoves structures that exist at the beginning of this stage would be:

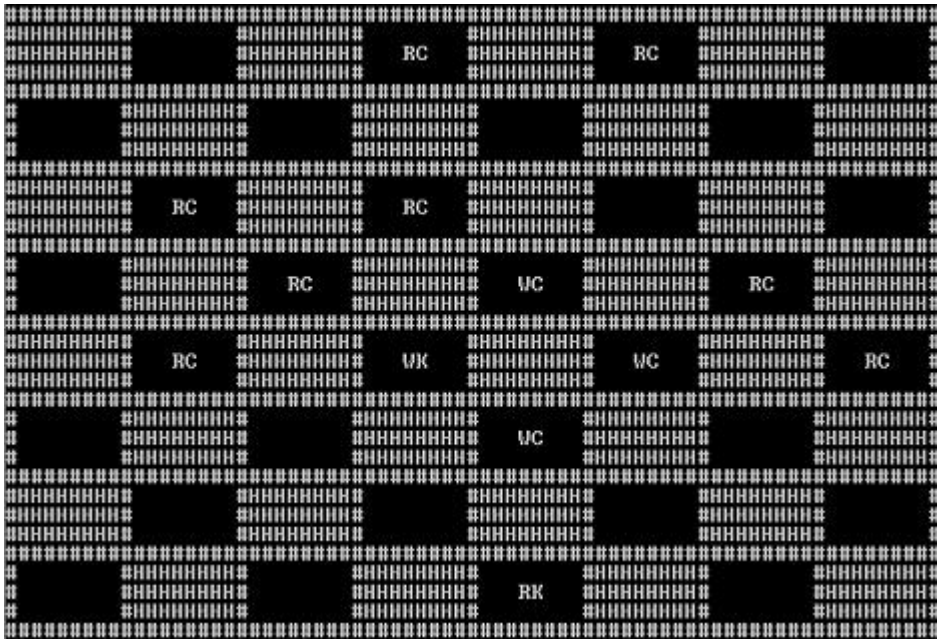

Figure 2.13: Example Board Position Where Some Pieces are "Helpless"

1. ^source 2, ^destination 6
2. ^source 2, ^destination 7
3. ^source 3, ^destination 7
4. ^source 3, ^destination 8
5. ^source 9, ^destination 13
6. ^source 17, ^destination 21
7. ^source 17, ^destination 22, ^riskAfterMove singleJump
8. ^source 20, ^destination 24
9. ^source 31, ^destination 26
10. ^source 31, ^destination 27

Unless otherwise noted, all these moves have both their ^riskBeforeMove and ^riskAfterMove set to "unKnown". In the case of the 7th move, its ^riskAfterMove is "singleJump" because of the white king present in square 18. It should also be noted that there would be two ^immobilePieceUnderThreat data structures corresponding to the pieces at squares 10 and 16. The piece at square 10 would have its ^savingSquare attribute set to 6 (since the white checker at square 15 would potentially "jump" it, and its destination would be square 6) while that at square 16 has its ^savingSquare set to 12.

2.3.5.3.2 Risk Analysis

The 1st move in the list above is from square 2 to square 6. It so happens that there exists an ^immobilePieceUnderThreat (the piece at square 10) with its ^savingSquare set to 6. Thus, making this move would result in saving that piece. In order to increase the

likelihood of this move being selected by the agent[8], the contents of this ^afterEvaluatedMoves structure is copied to a new one, its ^riskBeforeMove is set to "yes", ^riskAfterMove is set to "unKnown" and the original structure is deleted.

The 4th move in the list is from square 3 to square 8. Making this move would result in a double jump involving the immobile piece at square 16, and the piece that was just moved to square 8, since the ^savingSquare of the piece at square 16 is 12, which happens to lie in the same diagonal as 3 and 8, and is directly in front of 8. Thus, the corresponding ^afterEvaluatedMoves structure is copied to a new one, its ^riskBeforeMove is set to "unKnown", ^riskAfterMove is set to "doubleJump", and the original structure is deleted.

The 5th move in the list, from square 9 to square 13 looks harmless at first, but results in another piece getting jumped – specifically the piece at square 14. If the piece at square 9 is moved, the white king at square 18 would jump the piece at square 14. Hence, the corresponding ^afterEvaluatedMoves structure is copied to a new one, its ^riskBeforeMove is set to "unKnown", ^riskAfterMove set to "yes" (Though in this specific case, it results in a single jump, in other cases, it could lead to double or even triple jumps), and the original structure is deleted.

The 9th move in the list is from square 31 to square 26. While this move doesn't seem to have any apparent risks after move, making this move allows the opponent to move the white king located at square 18 to the empty square 22, placing both the checker at square 17, and the king that has now been moved to square 26 at risk simultaneously and thereby ensuring that the Red player loses at least one piece. Such situations usually arise at the latter stages of a game, and can have devastating consequences. Hence, the corresponding

[8] A more detailed explanation can be found in Section 2.3.8.

^afterEvaluatedMoves structure is copied to a new one, its ^riskBeforeMove is set to "unKnown", ^riskAfterMove is set to "doubleJump", and the old structure is deleted.

2.3.5.3.3 Changed Structures after Risk Analysis

Thus, the changes to the original list of ^afterEvaluatedMoves would be:

1. ^source 2, ^destination 6, ^riskBeforeMove yes, ^riskAfterMove unKnown
2. ^source 3, ^destination 8, ^riskBeforeMove unKnown, ^riskAfterMove doubleJump
3. ^source 9, ^destination 13, ^riskBeforeMove unKnown, ^riskAfterMove yes
4. ^source 31, ^destination 26, ^riskBeforeMove unKnown, ^riskAfterMove doubleJump

2.3.6 Trading Pieces

This is one area of Soar Checkers that needs to be improved. There are a lot of factors that need to be considered before it can be said if a given trade is "good" or not, but in general, all trades in Soar Checkers can be classified as:

2.3.6.1 Classification of Trades in Soar Checkers

- Even trade
- Good trade
- Bad trade
- Terrible trade

A trade is considered an Even trade when the same number of checkers are involved, for both players (for example, a red checker is traded for a white checker). A trade is called a Good trade when it gives the player initiating the trade some kind of advantage - this might be in terms of the number of pieces involved in the trade (for example, trading a red checker for two white checkers), the type of pieces involved in the trade (for example, trading a red checker for a white king), or the resulting board position (for example, a seemingly even trade which results in nullifying the positional advantage that the opponent might have had). A bad trade is one in which a king is traded for an opponent king, while a terrible trade is one in which a king is traded for a checker, or the number of pieces lost by the player during the trade is greater than the number of pieces lost by the opponent.

2.3.6.2 Good Trade Heuristics

A good checkers player usually trades pieces under the following circumstances:

- He or she is ahead in terms of the number of pieces, and its either an even, or a good trade.
- The trade results in either gaining a positional advantage, or weakens the opponent's position.

2.3.6.3 Implementation Difficulties

While these ideas seem simple, implementing them proved exceedingly difficult. For example, the rules that were required to encapsulate the various kinds of trades involving just one piece on each side numbered more than 75, and each of these rules involved checking the contents of a vast number of squares on the game board - all of which contributed to

slowing down the decision making process in Soar Checkers by an unacceptable level. Another stumbling block was the fact that under certain situations, the rules to evaluate trades resulted in changes to the working memory that went in direct conflict with the results of the earlier stages in the decision making process. These resulted in conditions known as operator-conflict impasses[9], which ultimately led to extremely poor decisions being made by the agent. Additionally, while it is fairly straightforward to say that a move which results in exchanging one piece for two is a good trade, it is infinitely more difficult to evaluate trades which result in positional advantages, since there really is no objective way to say with absolute confidence how "good" a board position is – unless huge game databases are used (which several world-class programs do, as explained in Chapter 1).

Due to the difficulties explained above, in this version of Soar Checkers, those moves that would result in pieces getting traded are avoided, given a choice. At the beginning of this stage, some of the ^afterEvaluatedMoves data structures at the top-state, with their ^riskAfterMove set to "unKnown" actually represent moves that would initiate the trading of pieces. Such moves are identified at this stage, the corresponding ^afterEvaluatedMoves data structures are copied to new ones, their ^riskAfterMove is set to "yes" (since making this move would result in getting at least single jumped, though the opponent piece involved in the jump would also potentially get jumped in the player's next move), and the original structures are removed from the top-state of the agent.

2.3.7 Picking a Move

Up to this point, the game state, as represented through the ^io.input-link of the Soar agent whose turn it is to move (denoted by having its ^isTurn attribute being set to 1), has

[9] Please refer to Appendix B for more details.

been analyzed to generate a list of moves, and add information to those moves regarding their respective risks or merits. At this stage of the decision making process, first, an operator called ^pickMove is proposed for each ^afterEvaluatedMoves data structure attached to the agent's top-state, and the corresponding data structure is copied to a new data structure called ^legalMoves and attached to the newly proposed operator. Then, one out of all the proposed ^pickMove operators is selected based on a set of strategies which would be explained in depth below. Finally, the selected ^pickMove operator is applied by copying the relevant portions of the ^legalMoves structure attached to it to the agent's ^io.output-link. The contents of the output-link thus represents the agent's move, are read by the game environment (using SML), and changes are made to the game board to reflect the resulting board position after making the move.

Let us take a more in-depth look at the operator selection process. As explained in Appendix B, an operator in Soar can have various "preferences" such as "indifferent", "better", "worse", "reject" etc associated with it. The goal being tried to accomplish here is to select the ^pickMove operator which has the "best" possible ^legalMoves data structure attached to it. This is done through a combination of rules which encode various strategies that the agent uses throughout the game. The most important strategies [11, 12] and techniques used are described below in no particular order.

2.3.8 Operator Selection Strategies

2.3.8.1 Reduce Risks - Prefer a move that reduces risk to all other moves.

In Soar Checkers, the game play is based on the idea of minimizing losses while maximizing gains. One of the first things that facilitates this is to consistently ensure that no pieces are lost as a result of a move, given a choice. Several scenarios need to be considered

to make this possible, but in essence, if there are two legalMoves data structures A and B such that both of them have their ^riskBeforeMove set to "yes", A has its ^riskAfterMove set to "unKnown" but B's ^riskAfterMove is set to a value other than unKnown (for example, "singleJump"), always prefer A over B. Closely related to this idea is avoiding moves that result in losses, given a choice. For example, if A and B are ^legalMoves data structures such that both have their ^riskBeforeMove set to "unKnown", A has its ^riskAfterMove set to "unKnown", but B's ^riskAfterMove is set to a value other than unKnown, prefer A over B since making move B would potentially result in losing a piece while A seems to be a safe move. Another idea that fits into the same theme is not to waste a move on a piece that is beyond rescuing. In other words, if all the ^legalMoves structures involving a given piece have their ^riskBeforeMove set to "yes", and none of them have their ^riskAfterMove set to "unKnown", it is best to pick a move involving a different piece that is safe, if it exists.

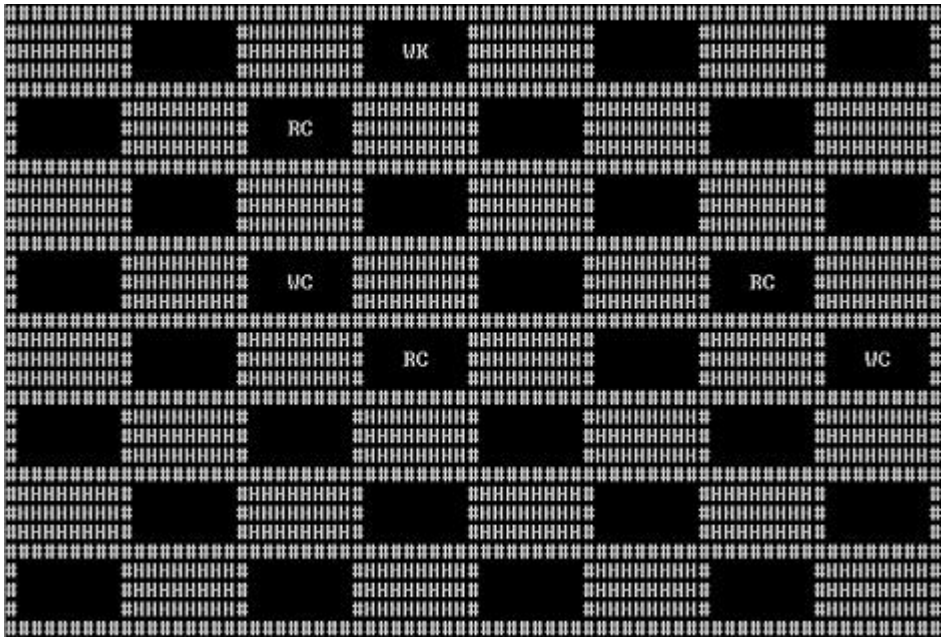

Figure 2.14: Example Board Position that Demonstrates the 'Reduce Risks' Strategy

Example - Consider the board position shown in Figure 2.14 above. Assuming that it is the Red agent's turn, the relevant portions of the ^legalMoves structures associated with the proposed ^pickMove operators would be:

1. ^source 6, ^destination 9, ^riskBeforeMove yes, ^riskAfterMove singleJump
2. ^source 6, ^destination 10, ^riskBeforeMove yes, ^riskAfterMove singleJump
3. ^source 16, ^destination 19, ^riskBeforeMove yes, ^riskAfterMove unKnown
4. ^source 18, ^destination 22, ^riskBeforeMove unKnown,
   ^riskAfterMove unKnown
5. ^source 18, ^destination 23, ^riskBeforeMove unKnown,
   ^riskAfterMove unKnown

The 1st and 2nd moves involve the red checker located at square 6, which is under threat by the white king occupying square 2. However, neither of the moves can save the checker from being jumped. Hence, the operator selection rules written for Soar Checkers would look for other moves that would either save a piece, or is not risky. Moves 4 and 5 involve the red checker located at square 18, which does not have any risks before making the move, and remains safe after either of the moves are made. Move 3 is of the most interest given this board position because in this case, the piece located at square 16 is under threat by the white checker at square 20, but once the move is made, the piece is safe. Obviously, moves 1 and 2 result in no perceivable gain, moves 4 and 5 are safe moves, but are overlooked in favor of move 3 because it results in saving a piece which could otherwise have been lost.

2.3.8.2 Thwart Opponent Crowning Moves - Given a choice, prevent the opponent checkers from getting crowned.

Motivation to Protect King Row - Not all squares on the checker board are created equal. The so-called "double corners" (the square closest to a player on his or her right hand side - square 1 if the player's pieces move from top to bottom, and square 32 if they move from bottom to top) are extremely useful while playing defensively. For example, given an endgame in which say Red has one king that occupies one of the double corners, and White has two kings, it takes about 40 moves in a precise sequence for White to win. However, in Soar Checkers, no opening game or endgame databases are used.

One of the key goals in defensive play is to prevent the opponent from crowning any of his or her checkers. A seemingly obvious way to achieve this end is to make sure that all the four squares in the "king row" (Assuming that Red plays from top to bottom, the king row for Red would have the squares 29-32 and that for White would have the squares 1-4) are occupied. The drawback of this strategy is that it hampers the player's ability to attack the opponent, or effectively protect his or her own pieces that move forward from being attacked. However, if the squares in the king row are left empty, it greatly increases the chances of the opponent successfully crowning a checker.

Plan of Action - A workable compromise between the two alternatives explained above is to occupy two specific squares in the opponent's king row – the player's double corner and the second legal square to its left. Assuming that Red plays from top to bottom, this means that the Red agent would try and not move its checkers from squares 1 and 3 while the White agent would keep its checkers in squares 30 and 32 for as long as possible. The benefits of this tactic are that two checkers are now free to assist the other checkers in either attack or defense, while the two checkers left in the king row can still prevent the

opponent's checkers from getting crowned, except when two of his or her checkers work in conjunction so that one of them can get crowned without the risk of getting jumped.

Example of Protecting the King Row - Let us take a look at two possible scenarios. Consider the board position shown in Figure 2.15. Assuming that it is the white player's turn, the possible moves are:

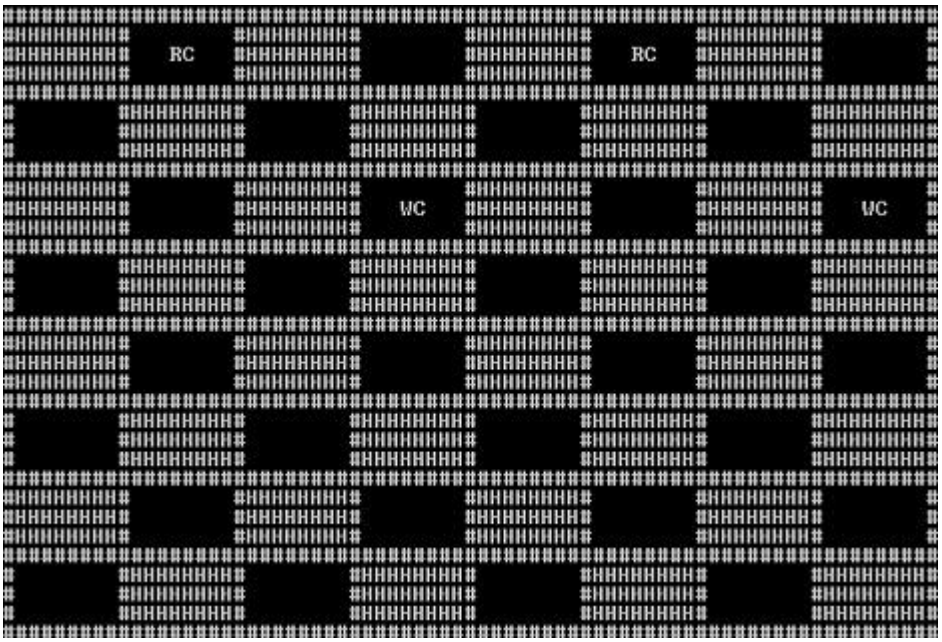

Figure 2.15: Example Board Position Demonstrating How to Protect the King Row with Only Two Checkers

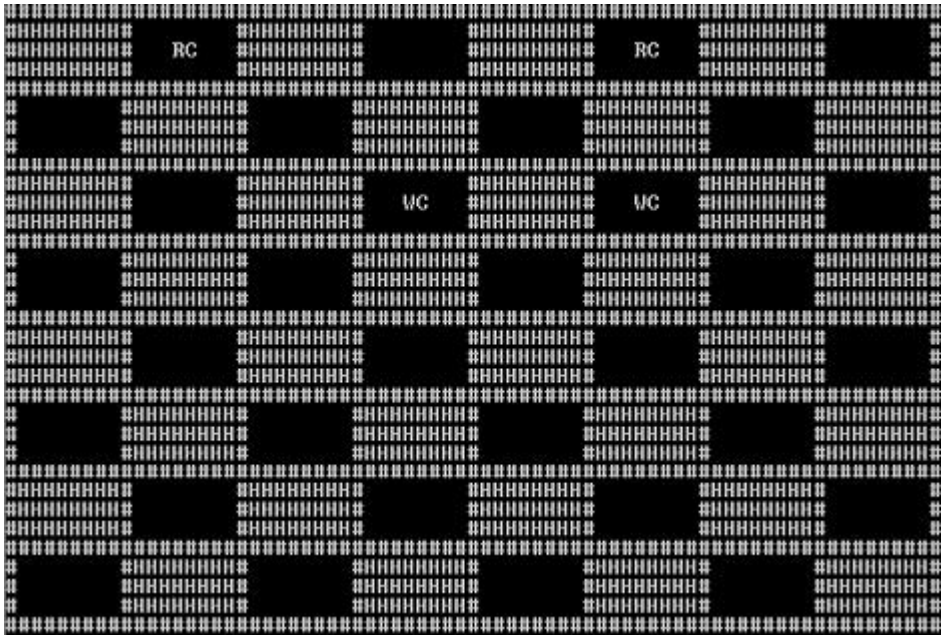

Figure 2.16: Example Board Position Demonstrating How to Get to the King Row with Two Checkers

1. ^source 10, ^destination 6, ^riskBeforeMove unKnown, ^riskAfterMove singleJump
2. ^source 10, ^destination 7, ^riskBeforeMove unKnown, ^riskAfterMove singleJump
3. ^source 12, ^destination 8, ^riskBeforeMove unKnown, ^riskAfterMove singleJump

As can be seen, though only two of the four possible squares in the king row are occupied, none of the moves White can make can lead to either of the white checkers getting crowned. Thus, this strategy is highly effective in most situations. However, a slightly different board position shows the fallibility of this strategy.

Example of Reaching the King Row - Consider the board position shown in Figure 2.16 above. Assuming that it is the White player's turn, these are the possible moves:

1. ^source 10, ^destination 6, ^riskBeforeMove unKnown, ^riskAfterMove singleJump
2. ^source 10, ^destination 7, ^riskBeforeMove unKnown, ^riskAfterMove singleJump
3. ^source 11, ^destination 7, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
4. ^source 11, ^destination 8, ^riskBeforeMove unKnown, ^riskAfterMove singleJump

The red checkers in squares 1 and 3 successfully thwart moves 1, 2 and 4, but move 3, in which the white checker is moved from square 11 to square 7 would indeed result in that checker getting crowned in white's next move, since it is protected from being jumped by the white checker on square 10. Thus, it is obvious that this strategy is not fool-proof when it comes to preventing an opponent checker from getting crowned. Still, it is used in Soar Checkers since in most situations, it is effective, and the benefits outweigh the costs.

2.3.8.3 Crown Checkers - Prefer a move that crowns a checker.

In checkers, a king is more powerful than a normal checker since it can move both forward and backward. Hence, it is desirable to convert checkers to kings by moving them forward till they reach the king row. In practice, everything else being equal, move A will be preferred over move B if A's destination is one of the squares in the player's king row.

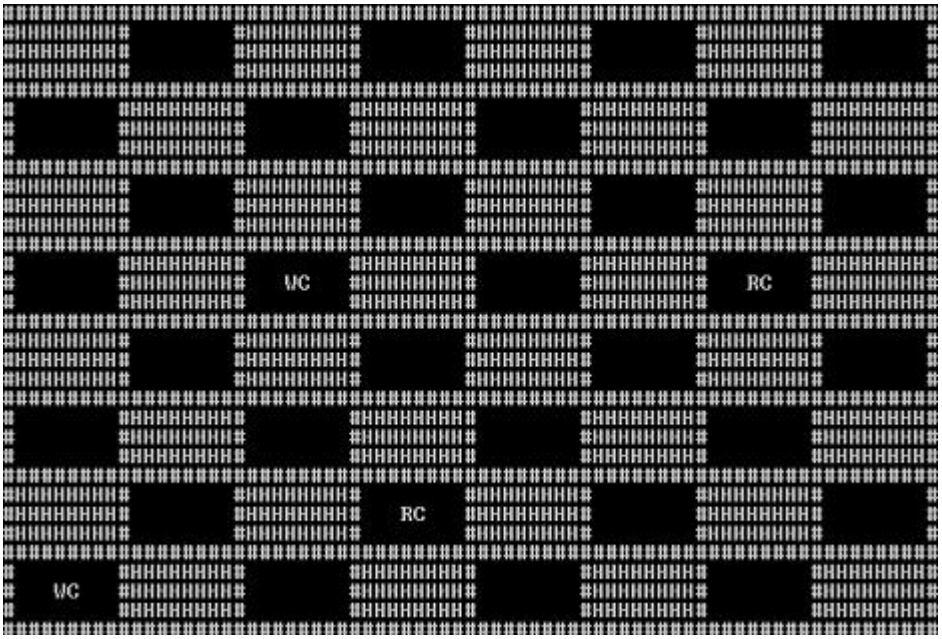

Figure 2.17: Example Board Position for a Red checker Being Crowned

Example - Consider the board position shown in Figure 2.17 above. Assuming that it is the Red player's turn to move, these are the relevant portions of the ^legalMoves data structures:

1. ^source 16, ^destination 19, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
2. ^source 16, ^destination 20, ^riskBeforeMove unKnown,
   ^riskAfterMove unKnown
3. ^source 26, ^destination 30, ^riskBeforeMove unKnown,
   ^riskAfterMove unKnown
4. ^source 26, ^destination 31, ^riskBeforeMove unKnown,
   ^riskAfterMove unKnown

In this case, all of the moves seem to be equal, since all of them have their ^riskBeforeMove and ^riskAfterMove set to "unKnown". However, moves 3 and 4 are considered superior to moves 1 and 2 in Soar Checkers, since both of them are risk-free and

result in a checker getting crowned. It should be noted that either move 3 or move 4 would get selected at random in this case.

2.3.8.4 Protect Kings - Given a choice, sacrifice a checker instead of a king.

During game play, certain situations arise when the opponent has gained a significant advantage, and leave you with no choice but to sacrifice one piece or the other in your next move. Since in checkers, a king is more powerful than an ordinary checker, it is usually good strategy to try and preserve your kings as much as possible. Thus, if there is a choice between sacrificing a checker, or sacrificing a king, in Soar Checkers, the agent would sacrifice the checker.

Example - Consider the board position shown in Figure 2.18. It is quite obvious that the Red player is in a losing position, and assuming that it is Red's turn to move, the relevant portions of the ^legalMoves data structures generated by the agent would be:

1. ^source 5, ^destination 9, ^riskBeforeMove unKnown, ^riskAfterMove singleJump
2. ^source 6, ^destination 9, ^riskBeforeMove unKnown, ^riskAfterMove singleJump
3. ^source 6, ^destination 10, ^riskBeforeMove unKnown, ^riskAfterMove singleJump
4. ^source 29, ^destination 25, ^riskBeforeMove unKnown,
   ^riskAfterMove singleJump

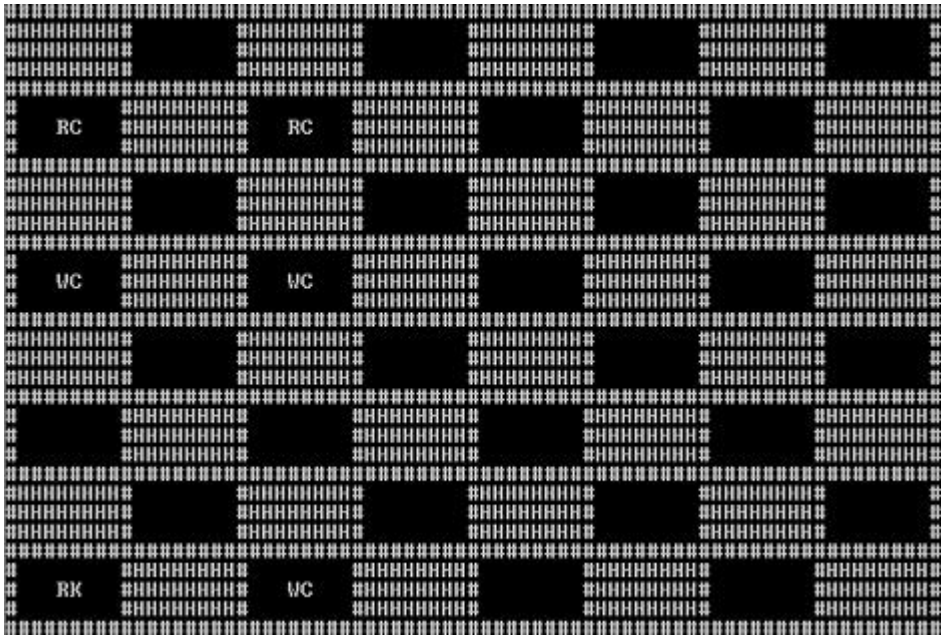

Figure 2.18: Example Board Position for Sacrificing a Checker to Protect a King

It can be seen that all four available legal moves result in a piece getting jumped. However, the pieces at squares 5 and 6 are red checkers while square 29 has a red king. In this case, Soar Checkers would pick a move that results in one of the checkers getting jumped. It can be seen that given the board position, the "best" move is move 1, since this would force the white checker at square 14 to jump the red checker that was just moved to square 9, thereby allowing the Red player to move the checker at square 6 to square 10 safely in his or her next move. All the other moves result in the game being over within the next 2 moves for each player.

2.3.8.5 Minimize Material Losses - If getting jumped is inevitable, pick the move that minimizes the number of pieces lost.

Sometimes, during the course of a game, a player is confronted with choosing between two or more moves that result in pieces getting jumped, but the number of pieces lost as a

result of those moves might not be the same. In Soar Checkers, there are three values pertaining to the risk of getting jumped, that might be assigned to the ^riskAfterMove attribute of a ^legalMoves data structure:

potentialSingleJump

singleJump

doubleJump

Given a choice among moves with varying degrees of risk of being jumped, the order of preference in Soar Checkers is:

doubleJump < singleJump < potentialSingleJump

Example - This will be illustrated in the following example. Consider the board position shown in Figure 2.19 on the next page. Assuming that it is the Red player's turn to move, the relevant portions of the ^legalMoves data structures generated by the agent would be:

1. ^source 7, ^destination 10, ^riskBeforeMove unKnown,
   ^riskAfterMove potentialSingleJump

2. ^source 11, ^destination 15, ^riskBeforeMove unKnown,
   ^riskAfterMove doubleJump

3. ^source 11, ^destination 16, ^riskBeforeMove unKnown,
   ^riskAfterMove doubleJump

4. ^source 21, ^destination 25, ^riskBeforeMove unKnown,
   ^riskAfterMove singleJump

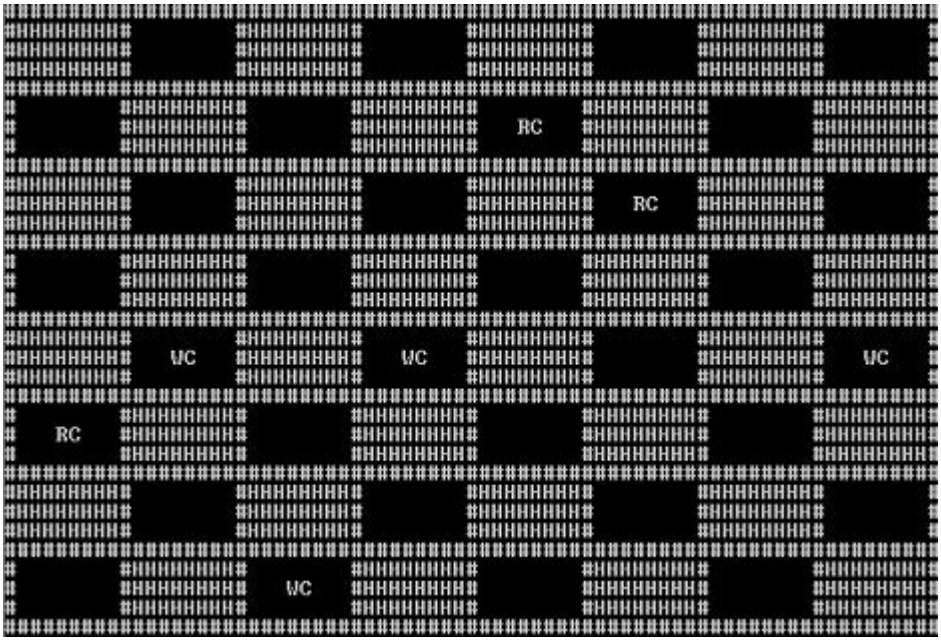

Figure 2.19: Example Board Position Demonstrating 'Minimize Material Losses' Strategy

Move 1 has a ^riskAfterMove of "potentialSingleJump" because the White player could move the white checker from square 18 to square 14, thereby putting the red checker that had just been moved to square 10 under threat of being jumped. However, in this case, the red checker can avoid getting jumped by moving to square 15, which would now be a safe move.

Move 2 has a ^riskAfterMove value of "doubleJump" because the white checker occupying square 18 would jump both the red checker that was just moved to square 15, and the red checker in square 7. Thus, if there is any other option, Soar Checkers would not make this move.

Move 3, quite similar to move 2 results in a double jump except that this time, it is because of the white checker occupying square 20. Once again, this would be among the last choice moves for the agent.

Finally, move 4 results in the red checker at square 21 getting jumped by the white checker at square 30. After considering all options available, the agent would pick move 1, which is indeed the best possible move, given the board position.

2.3.8.6 Occupy Center Squares - Prefer a move towards a center square over one towards an edge.

It is sound checkers strategy to try and occupy the squares in the center of the board. The reason for this is that, if a piece is placed in a square on an edge or a corner, its mobility is severely limited, and that piece can be easily trapped by the opponent. On the other hand, a piece located in a square that is away from the edges has more options to move forward - either in attack, or to defend another piece that might be under threat. In most games in checkers, the player whose pieces control the centre squares usually also has control over the game, and is most likely to win. Due to these reasons, in Soar Checkers, the agent prefers moves that have their destination as edge, or corner squares (with the exception of moves to the king row, which would result in crowning a checker) less than moves with other destination squares.

Example - Consider the board position shown in Figure 2.20 on the following page. Assuming that it is the Red player's turn to move, the relevant portions of the ^legalMoves data structures generated by the agent would be:

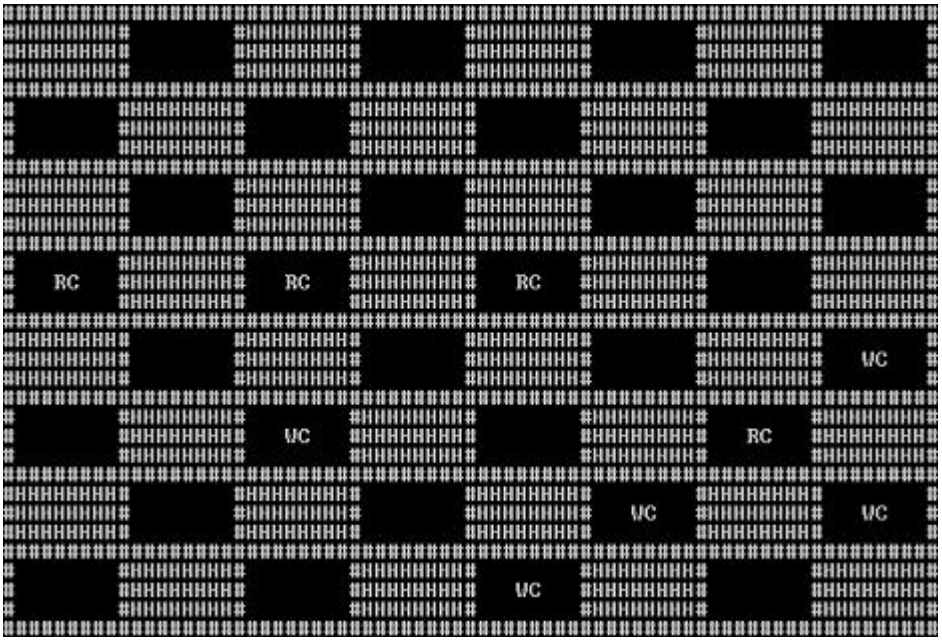

Figure 2.20: Example Board Position Demonstrating Good Positional Play

1. ^source 13, ^destination 17, ^riskBeforeMove unKnown, ^riskAfterMove yes
2. ^source 14, ^destination 17, ^riskBeforeMove unKnown,
   ^riskAfterMove unKnown, ^tradeType evenTrade
3. ^source 14, ^destination 18, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
4. ^source 15, ^destination 18, ^riskBeforeMove yes, ^riskAfterMove singleJump
5. ^source 15, ^destination 19, ^riskBeforeMove yes, ^riskAfterMove unknown

Analysis - Square 13 is on the edge of the board, and a piece occupying it can move in only one direction, making it relatively easy to trap it. That is indeed the case here, as can be seen in move 1, when the red checker in square 13 will be jumped by the white checker in square 22, if it is moved to the only square it can be moved to, 17.

Squares 14 and 15 are not edge squares, and hence, automatically give pieces occupying them more options when it comes to moving. In move 2, the red checker in square 14 can

be used to attack the white checker in square 22. However, this could potentially result in a trade of checkers, since the white checker occupying square 31 would jump the red checker that would end up in square 26 as a result of jumping the piece in square 22. This is reflected in the ^legalMoves structure with the attribute-value pair of ^tradeType evenTrade.

A safer option to attack the white checker in square 22 is move 3, and would get picked by the agent if there are no others moves with a higher preference. It can be seen that making that move would certainly result in the loss of the white checker at square 22.

Moves 4 and 5 involve the red checker at square 15, and both have a ^riskBeforeMove value of "yes". This is because the red checker at square 24 is under threat of being jumped by the white checker at square 28, and since square 15 happens to be in the same diagonal line and the square behind it in the same diagonal - square 10 - is empty, a double jump would result in which the red checker at squares 15 and 24 would be lost. The 4th move would result in the red checker that has now been moved to square 18 getting jumped by the white checker at square 22, and hence, is not a good option.

The 5th and final move involves moving the red checker from square 15 to square 19, and is the best move under the circumstances since it prevents a double jump and saves the red checker at square 24 that was under threat. As has been explained before, Soar Checkers gives the highest preference to a move that actually reduces risk and hence, move 5 will be picked by the Soar agent.

It can be seen from this example that occupying squares that are more towards the center of the board, and away from the edges is usually a good idea because it gives the player more options to attack (moves 2 and 3) or defend (move 5), and decreases the likelihood of a piece getting trapped (move 1).

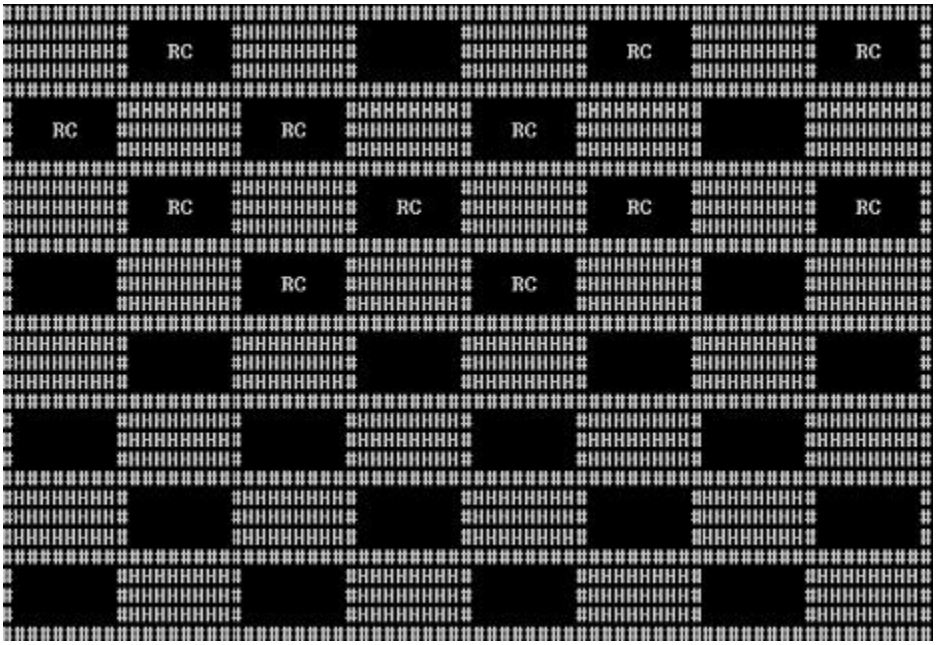

Figure 2.21: Example Board Position Showing the Wedge Formation

2.3.8.7 Wedge Formation - Create a wedge formation, built up from the back [12].

Motivation - The overall strategy to play winning checkers is quite simple – move your pieces forward while preserving them as much as possible, convert them to kings, and attack the remaining pieces of the opponent. If a piece is pushed forward, while it is closer to becoming a king, there is also an increased likelihood that it is vulnerable to being attacked by the opponent. However, if a piece is moved forward after ensuring that it has adequate back-up, it is much safer since the back-up pieces could thwart any potential attacks on it.

In Soar Checkers, an agent prefers moves that create a wedge-shaped formation, built up from the back row, with the exception of the two pieces that are kept in the back-row for as long as possible, to prevent the opponent from crowning one of his or her checkers. The benefits of this tactic have been explained above, and would be more apparent from the following example.

Example - Consider the board position shown in Figure 2.21 on the preceding page. The wedge-shaped formation can be seen in two instances - the first one is formed by the red checkers in squares 9, 10, 14 and the second one is formed by the pieces in the squares 10, 11 and 15. From the initial board position, the first wedge formation could have been obtained by first moving the checker from square 9 to square 14, then moving the checker from square 6 to square 9, and finally moving the checker from square 2 to square 6. It should be noted that another properly backed-up wedge could have been created by moving the checker from square 1 to square 6, but this move is not picked because the agent tries to keep squares 1 and 3 occupied for as long as possible, to safeguard the king row. Assuming that it is the Red player's turn, the relevant portions of the ^legalMoves data structures would be:

1. ^source 3, ^destination 8, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
2. ^source 4, ^destination 8, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
3. ^source 9, ^destination 13, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
4. ^source 12, ^destination 16, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
5. ^source 14, ^destination 17, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
6. ^source 14, ^destination 18, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
7. ^source 15, ^destination 18, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
8. ^source 15, ^destination 19, ^riskBeforeMove unKnown, ^riskAfterMove unknown

In this case, all the available moves are equivalent, in terms of risks before making the move, and risks after. However, move 3 will be the last choice because this involves moving to a square on the edge of the board, square 13. Moves 4, 5, 6, 7 and 8 will be overlooked

because the agent tries to build up a wedge formation, starting from behind, so that each piece in front is properly backed up. This leaves moves 1 and 2, both of which are from the back row, and succeed in building a well backed-up wedge. However, move 1 involves moving a checker from square 3, which the agent would not do unless there is no other safe move, which is not the case here. Thus, the agent would pick move 2, which involves moving the red checker from square 4 to square 8, to fully back up the wedge formed by the checkers in squares 10, 11 and 15.

2.3.8.8 Attack with Kings

This is arguably, the most complex portion of Soar Checkers and a significant amount of the logic involved is encoded within the game environment in C++ in the form of helper functions (as can be seen in Figure 2.11 on page 38). One of the attributes present on the Soar agent's ^io.input-link is called "^attackingMove". This attribute has two sub-attributes called "^attackingSource" and "^attackingDestination" that hold integer values representing locations of squares on the game board. Until one of the agent's checkers is crowned, both these attributes will be set to "-1", and are not used by the agent to make decisions at all.

Classification of Moves - There are three kinds of moves that the game environment considers, and depending on various factors, one of them is chosen, and the source of that move is set as the value for ^attackingSource and the destination of that move is set as the value for ^attackingDestination. The three types of moves considered are:

1. Chase Move
2. Trap Move
3. Attack Move

The game environment keeps track of the last move made by the opponent, and tries to determine if there exists a path on the game board that is relatively free of risks, and allows a king to chase the piece that was last moved by the opponent. This strategy is highly useful when the game is fairly advanced, and the opponent is trying to have his or her checkers crowned. An added advantage here is that while the king is trying to chase the piece that was last moved by the opponent, it might encounter other opponent pieces on the way, possibly placing them under threat too. The game environment uses the well known Djisktra's path-finding algorithm to compute paths on the game board that are then used to find all three types of moves. The shortest path that satisfies the qualifying criteria for each type of move is always used, and the number of hops required is considered as the measure of distance.

Example for Chase Moves - Consider the board position shown in Figure 2.22. White's last move was to move the white checker located at square 24 to square 19. This information is encoded in an attribute called "^hisLastMove" on the Red agent's ^io.input-link, which has sub-attributes called "^hisLastSource" and "^hisLastDestination" which would have values of 24 and 19 respectively in this case. Assuming that it is the Red agent's turn to move, the relevant portions of the ^legalMoves data structures would be:

1. ^source 1, ^destination 5, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
2. ^source 1, ^destination 6, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
3. ^source 3, ^destination 7, ^riskBeforeMove unKnown, ^riskAfterMove unknown

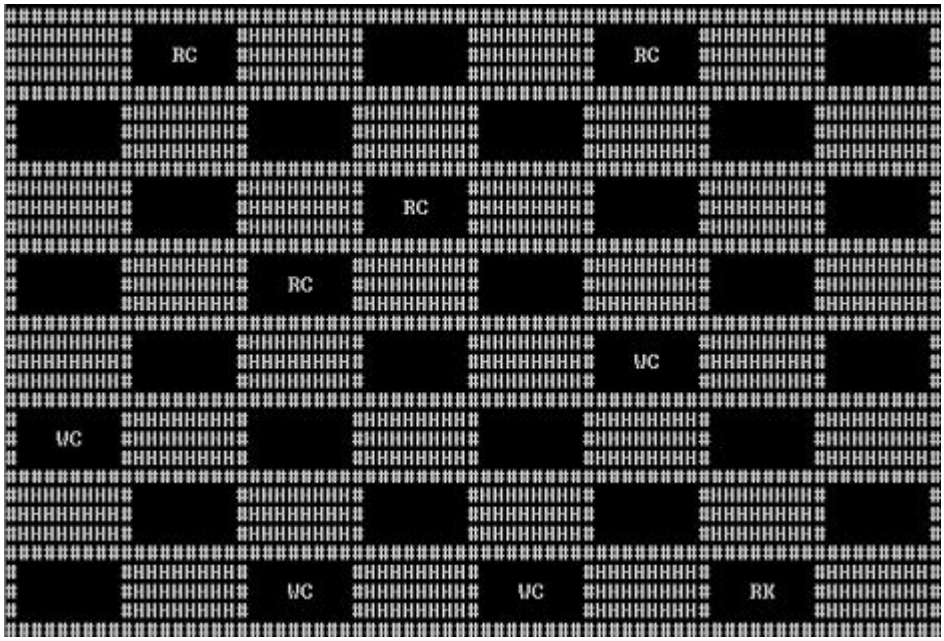

Figure 2.22: Example Board Position Demonstrating a 'Chase Move'

4. ^source 3, ^destination 8, ^riskBeforeMove unKnown, ^riskAfterMove unKnown

5. ^source 10, ^destination 15, ^riskBeforeMove unKnown,
   ^riskAfterMove singleJump

6. ^source 14, ^destination 17, ^riskBeforeMove unKnown,
   ^riskAfterMove doubleJump

7. ^source 14, ^destination 18, ^riskBeforeMove unKnown, ^riskAfterMove unKnown

8. ^source 32, ^destination 27, ^riskBeforeMove unKnown, ^riskAfterMove yes

9. ^source 32, ^destination 28, ^riskBeforeMove unKnown, ^riskAfterMove unknown

Analysis - Moves 1 through 4 are automatically the last choice since they involve moving the checkers from squares 1 or 3, thereby exposing the king row to the enemy. Moves 5 and 6 are automatically rejected since they actually result in losses (single jump in the case of move 5, and double jump in the case of move 6). Similarly, move 8 would result in the red king that was just moved to square 27 getting jumped by the white checker in

square 31 and hence, will be avoided if there is a choice. That leaves moves 7 and 9, both of which are equivalent in terms of their risks before and after making the move.

Since Red has a king, the game environment would try and find a move that would help the king chase the piece that was last moved by the opponent safely. In this case, the closest squares to square 19 (the destination of White's last move) from square 32 are squares 23 and 24. The path-finding algorithm will come up with three possible paths to these squares:

1. 32 -> 27 -> 23
2. 32 -> 27 -> 24
3. 32 -> 28 -> 24

Paths 1 and 2 are not suitable since both of them would result in the king being jumped by the white checker at square 31. Hence, the remaining path is chosen by the environment, and the first move in the path, from square 32 to square 28, is recommended to the Soar agent by setting the ^attackingSource and ^attackingDestination attributes on its ^io.input-link.attackingMove to 32 and 28 respectively. The preference rules for the agent are written in such a way that if there exists a valid ^attackingMove (that is, the ^attackingSource and ^attackingDestination attributes are not set to -1) that has a ^riskAfterMove of unKnown, that move would be preferred over almost all other alternative moves - some of the exceptions being moves that reduce risk (having a ^riskBeforeMove of "yes" and ^riskAfterMove of "unKnown"), moves that result in crowning a checker etc. Thus, in this case, move 9 is the recommended attacking move, and will be preferred over move 7. If the environment cannot find a safe chase move, it will try to find a safe trap move, which will be explained next.

Trap Moves - In Soar Checkers, a move that results in restricting the mobility of an opponent piece is considered a trap move. In the absence of a suitable, safe chase move, the game environment tries to find a move that would result in trapping one or more pieces of the opponent. Unlike a chase move, a trap move could involve checkers as well as kings. While calculating trap paths, care is taken to ensure that pieces that have already trapped an opponent piece are considered last, so that the chances of making a trap move that would result in freeing an already trapped opponent piece are minimized. The safest trap path which is also the shortest distance away from the target square is selected, and the first move in the path is recommended to the agent through the ^attackingMove attribute on it's ^io.input-link.

Example - Consider the board position shown in Figure 2.23 on the next page. Assuming that it is the Red agent's turn to move, the relevant portions of the ^legalMoves data structures would be:

1. ^source 1, ^destination 5, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
2. ^source 1, ^destination 6, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
3. ^source 3, ^destination 7, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
4. ^source 3, ^destination 8, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
5. ^source 10, ^destination 14, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
6. ^source 10, ^destination 15, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
7. ^source 32, ^destination 27, ^riskBeforeMove unKnown, ^riskAfterMove unknown

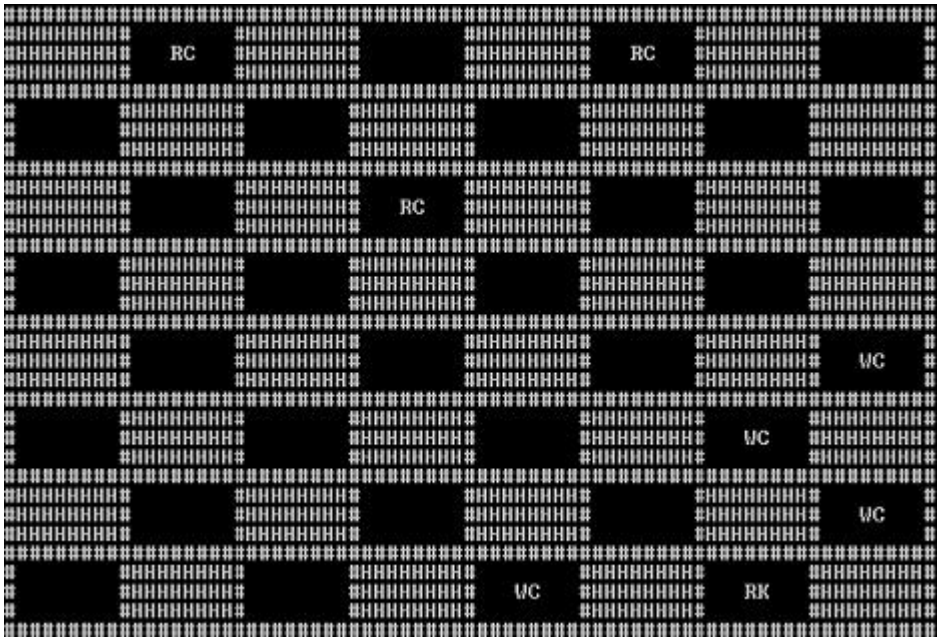

Figure 2.23: Example Board Position Demonstrating a 'Trap Move'

Analysis - Moves 1 through 4 are of the last priority for the agent, since they involve moving pieces from the opponent's king row square, thereby increasing the chances of an opponent checker getting crowned. That leaves moves 5 through 7, all of which are equivalent in terms of their risks before and after making the moves. In this case, the game environment would come up with 3 possible destination squares to trap opponent pieces - squares 11, 15 and 23. This is because placing a Red piece in square 11 would make the only move available to the white checker at square 20, which is to move to square 16, unsafe. Similarly, placing a Red piece in square 15 would trap the white checker at square 24, and placing a Red piece in square 23 would trap the white checker at square 31. Next, the game environment calculates the shortest paths to each of these possible destination squares, and in this case, they would be:

1. 3 -> 8 -> 11 (or 3 -> 7 -> 11)
2. 10 -> 15
3. 32 -> 27 -> 23

The first path would automatically be rejected since it involves moving from the opponent's king row. Even otherwise, among the 3 candidate paths, the shortest path would be selected, which, in this case is path 2 which involves a single move and would trap the white checker at square 24. Thus, among moves 5 through 7, move 6 would be selected by the game environment, and recommended to the agent through its ^io.input-link.attackingMove attribute.

Attack Moves - If no trap move can be found because all of the opponent's pieces are already trapped, the game environment tries to find a safe "attack move". An attack move is one which basically "goes in for the kill", or places an already trapped opponent piece directly under threat while ensuring that making that move does not result in opening up an avenue for that piece to escape. Like chase and trap moves, if such a move is found, the game environment recommends it to the Soar agent through the ^attackingMove attribute on its ^io.input-link.

Example - Consider the board position shown in Figure 2.24 on the next page. Assuming that it is the Red agent's turn to move, the relevant portions of the ^legalMoves data structures would be:

1. ^source 1, ^destination 5, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
2. ^source 1, ^destination 6, ^riskBeforeMove unKnown, ^riskAfterMove unKnown
3. ^source 3, ^destination 8, ^riskBeforeMove unKnown, ^riskAfterMove unknown

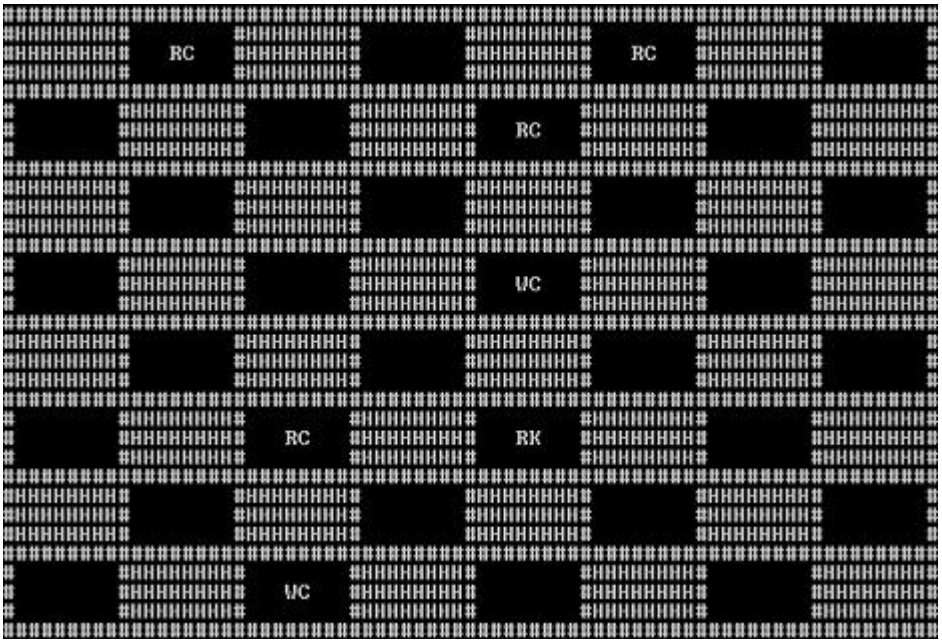

Figure 2.24: Example Board Position Demonstrating an 'Attack Move'

4. ^source 7, ^destination 10, ^riskBeforeMove unKnown,
   ^riskAfterMove yes, ^tradeType evenTrade

5. ^source 7, ^destination 11, ^riskBeforeMove unKnown,
   ^riskAfterMove yes, ^tradeType evenTrade

6. ^source 22, ^destination 25, ^riskBeforeMove unKnown,
   ^riskAfterMove singleJump

7. ^source 22, ^destination 26, ^riskBeforeMove unKnown, ^riskAfterMove unKnown

8. ^source 23, ^destination 18, ^riskBeforeMove unKnown, ^riskAfterMove unKnown

9. ^source 23, ^destination 19, ^riskBeforeMove unKnown, ^riskAfterMove unknown

Analysis - Moves 1 through 3 would be considered last since they would involve moving pieces from the opponent's king row. Moves 4 and 5 will not be considered since they have a ^riskAfterMove value that is not "unKnown" - both the moves result in a single trade of the Red checker at square 7 for the White checker at square 15, and this is reflected

in their ^tradeType value of "evenTrade". Move 6 will be rejected since it would involve an uncompensated loss of the Red checker at square 22. Moves 7 through 9 are equivalent in terms of their risks before and after making the moves. A careful analysis of the board position would show that the Red checker at square 7 has trapped the White checker at square 15 by making both its available moves (to squares 10 and 11) risky, while the Red checker at square 22 and the Red king at square 23 similarly have trapped the White checker at square 30. It should be noted that if move 7 is selected, it would give White a chance to prolong the game (White's position is so weak that its highly unlikely that it would avoid being beaten) by moving the White checker at square 30 to square 25. However, it can be noted that the Red king can be moved from square 23 while still having the White checker at square 30 trapped by the checker at square 22. Moves 8 and 9 do exactly this, and both are attacking moves, since they place the White checker at square 15 in immediate danger of being jumped while leaving all the opponent pieces that were previously trapped, still trapped. Thus, the environment would select one out of moves 8 and 9 randomly and recommend it as the ^attackingMove to the agent.

2.3.8.9 Avoid Repeating Moves – Provided there is another safe choice.

As the game was developed, it was noticed that more often than not, while using the king to chase or attack, one undesirable side effect was that the agent would move the king from square A to square B in move 1, and then, in move 2, would move the king that is now in square B back to square A, thereby effectively wasting two moves. To avoid this, a new attribute called ^myLastMove with sub-attributes called ^myLastSource and ^myLastDestination were added to the ^io.input-link of the agent which stored the values of the agent's previous move's source and destination squares respectively. While deciding

which move to pick from the available collection of ^legalMoves data structures, one which has its ^source the same as ^io.input-link.myLastMove.myLastDestination and ^destination the same as ^io.input-link.myLastMove.myLastSource would be made the last choice unless it was recommended as the ^attackingMove, or there are no other safe alternatives.

2.3.9 Picking a Jump

Quite similar to how in the case of normal moves, the list of available moves are finally represented in terms of ^legalMoves data structures attached to the top-state, the list of available jumps are represented in terms of ^legalJumps data structures. Each of these data structures have an attribute called "^numJumps" whose value indicates how many opponent pieces would be jumped as a result of this particular jump (1 in the case of single jump, 2 in the case of double jump, etc.) and depending on its value, various operators such as ^pickSingleJump, ^pickDoubleJump, ^pickTripleJump etc are proposed and the ^legalJumps data structure is copied and attached to the newly proposed operator.

Preference Rules for Jumps - The copied ^legalJumps data structures contain information about what kind of opponent pieces would be jumped over, and this is used, along with the number of pieces involved in a jump to pick a jump from the various possible jumps (if such a case exists). These preference rules can be summarized as:

- A single jump is preferred over a single move.
- A single jump over an opponent king is preferred over a single jump over an opponent checker.
- A double jump is preferred over a single jump.
- A double jump which results in the opponent losing at least one king is preferred over one which does not.

- A triple jump is preferred over a double jump.
- A triple jump that results in gaining at least one opponent king is preferred over one that does not.

It should be noted that in the case of jumps, no risk analysis is performed, which is not the case for normal moves. This is partly because the rules of checkers require that if a jump can be made, it must be made. However, when there are multiple options for jumps, some options might be "better" than the others and some of the ideas used to pick a move can be modified slightly to aid in deciding which jump to pick among available jumps which seem to be equivalent in terms of the number of pieces gained, and the kinds of pieces of gained. On the other hand, such situations do not arise too often and so, it was decided that any gains as far as better game-play is concerned are offset by the development time and effort.

2.3.10 Output and Cleaning up

At the end of the decision making process, if a jump is available, one of the available jumps would be selected, and the relevant portions of the data structure that is used to represent it (a copy of its corresponding ^legalJumps) would be written to the ^io.output-link, which would then be read by the game environment (using SML) to update the game board accordingly. Similarly, if no jumps are possible, one of the available moves is selected, and the relevant portions of the corresponding ^legalMoves data structure are written to the ^io.output-link of the agent.

The next stage before reverting to the stage where all possible moves and jumps are generated is to clean up all the remaining ^legalMoves and possibly, ^legalJumps data structures that were generated as part of the agent's decision cycle, along with resetting certain attributes attached to the top-state that are used by the agent to propose the correct

operator at the correct time. All this is done by an operator called ^cleanUpLegalMovesAndJumps.

Finally, the game ends when one of the agents involved cannot generate any moves or jumps, either because it does not have any pieces left, or all of its remaining pieces are completely trapped. In this case, the agent puts an attribute called ^numJumps on its ^io.output-link with value 0 and the game environment would declare the player or agent with the opposite color as the winner and terminate the game.

CHAPTER THREE
RESULTS

3.1 Review of Objective

The problem being attempted to solve was to see if it is possible to build an intelligent agent that could play checkers at a fairly decent level without using the traditional computer chess or checkers methods described in Chapter One (minimax algorithm, alpha-beta pruning, iterative deepening etc to name a few). An attempt was made to construct a rules-based expert system that relies almost entirely on generalizable strategies to exhibit its "intelligent" game-play and would be good enough to beat novices fairly consistently and at least challenge players at the "advanced novice" to intermediate level. Soar was chosen as the platform for building this agent because it came with built-in features that facilitate creating rules-based expert systems, has been proven to be fairly reliable in developing such systems (including flight simulators) for over twenty years and makes it relatively straight-forward to have multiple agents play each other, and to add or modify features or strategies to the agents.

3.2 Assessment

The problem definition makes it inherently hard to quantify the results objectively. The objectives stated above were met successfully for the most part though it is felt that there is still some way to go before Soar Checkers can be declared an unqualified success. The results of this experiment can be best summarized as follows:

The agent, after continuous refinement, was able to play at a level good enough to defeat or achieve draws against human opponents who were either novices, or what can be described as "advanced novices" (in other words, they are not novices, but don't play at a level high enough to be classed intermediate) on a fairly regular basis. Some of these opponents played the agent directly, while other matches were played against opponents online who had rated themselves as being novices or advanced novices.

Since the agent picks a random move out of the available legal moves or jumps (subject to certain preferences, according to strategy, of course), its game-play is fairly unpredictable, from game to game. This feature causes one major disadvantage - it is almost impossible to predict if the agent will consistently beat a certain opponent. In checkers, certain openings are known losses, while other seemingly harmless positions that can be reached within four moves of the start of the game have also been proven to be forced losses [1] - meaning even with perfect play from there on, the agent cannot prevent a defeat. The problem with random moves combined with strategy is that some of these moves, especially in the opening phase, are seemingly strategically sound (for example, prefer a move that does not result in losing a piece, over one that results in a piece being sacrificed, with no tangible gain), but lead to the aforementioned positions resulting in forced losses.

3.2.1 Comparison with Chinook

There is an online version of Chinook [13], which can be challenged by opponents the world over. It uses only a single processor and some of the search extensions used by the tournament version of Chinook are disabled, but has the complete and perfect six-piece endgame database, and the opening book from 1990. While this is certainly a weaker version of Chinook than the one used in tournaments, it is still a formidable opponent. It even plays

at different levels such as novice, intermediate and advanced, with the difference being the amount of time it takes to return a move (in other words, how deep it searches the minimax game tree using alpha-beta pruning). While a game is in progress, it uses its position evaluation function to analyze the state of the game, and displays comments for the reader's benefit, such as "Chinook has a small advantage ".

3.2.1.1 Effect of the Lack of an Opening Book

It was found that even at novice level, Chinook is far stronger than Soar Checkers. While this is a disappointing result, it shouldn't be a big surprise. The lack of an opening book in Soar Checkers severely hampers it against any opponent (human or machine) that has invested time in learning or storing openings. As explained before, unlike in chess, where a weak opening line may result in a player entering the middle game phase with a material disadvantage, or an inferior position, in checkers, a mistake made during the opening game is almost always fatal, and cannot be recovered from, even with perfect play. It is impossible to play correctly every time, especially during the opening game using pure strategy, since there are so many situations or board positions in checkers for which it has taken decades of human analysis to decide on the "right" move that will at least prevent an otherwise forced loss. For example, in what is known as the "White Doctor" opening, many decades of analysis has resulted in the conclusion that the correct play requires one side to sacrifice a piece (which runs directly against one of the basic strategies to play good checkers - to pick a move that is "safe" over one that results in an uncompensated loss) [1].

Thus, though the online version of Chinook is weaker than its tournament version, it still uses an opening book that is known to have approximately 2000 different positions (they actually use an anti-book, or a book of positions not to get into, as explained in Chapter

One). It can be easily seen that Soar Checkers is almost certain to make moves that would result in forced losses, in the absence of any opening book or database, thus making it susceptible to losing against Chinook, or even human opponents who are familiar with book play and documented opening traps.

3.2.1.2 Importance of Endgame Database

Among multiple games played against Chinook, in those instances where Soar Checkers did not make an inadvertent opening blunder and wind up in a losing position, it was seen that it played a respectable middle game. This shows that strategies can indeed contribute to decent game play, at least in the middle game phase of the game. However, once the number of pieces had decreased sufficiently to enable Chinook's endgame database to kick in, Soar Checkers was no match for Chinook. Once again, this demonstrates the fact that an agent that uses generalized strategies for its game play might be competitive against human players at the novice or advanced novice level, but against an opponent that has perfect information in the endgame (machine or human), it does not stand a realistic chance. It is worth mentioning that the Chinook team developed a six-piece endgame database because it was observed by them that the human players at the Masters level seemed to have perfect knowledge of the endgame involving up to five different pieces. Thus, by building the six-piece database, they were able to make moves with absolute certainty of being perfect in the endgame, while the human opponent could at best, make an educated guess.

3.2.2 Comparison with Another Program

3.2.2.1 Soar versus Computer

There is an online checkers program at http://www.nabiscoworld.com which provides a couple of options - the user could either play against a human opponent, or choose to play against the computer. While playing against the computer, it allows the user to select the difficulty level, ranging from one to ten, with one being the easiest and ten, the hardest. Soar Checkers was pitted against this program at different difficulty levels, and it was found that it beat the program with ease, and quite regularly, at difficulty levels ranging from one to six. At levels seven and eight, it was more of a challenge to beat the program, but Soar Checkers still emerged victorious about 50% of the time. Levels nine and ten proved much harder to beat, as evidently, at these levels the program would be searching the deepest to come up with its moves, and would also be using other techniques mentioned in Chapter One. The problem here though, is that once again, it is hard to quantify these results, since the strength of the checkers program used by this website is not known.

3.2.2.2 Soar versus Humans

Soar Checkers was also used to play against human opponents who were using the same website, and as expected, against opponents who were novices, it won almost all the time, barring cases where it made mistakes in the opening game, and ended up in positions that were forced losses. It was seen that against opponents who were familiar with endgames, Soar Checkers seemed to fritter away advantages that were gained during the opening and middlegame phases of the game. For example, in one game, while playing White, Soar Checkers went into the endgame with two kings and four checkers, while the human opponent had only one king and three checkers, two of which were tied up. An

accomplished player would certainly have gone on to win from this position with precise, calculated moves. Unfortunately, relying on pure strategy is not sufficient in the endgame, especially against opponents who are not novices, and in this specific case, the human opponent managed to escape with a draw. The recreated board position with Soar Checkers having a huge advantage while going into the endgame, playing White, can be seen in Figure 3.1.

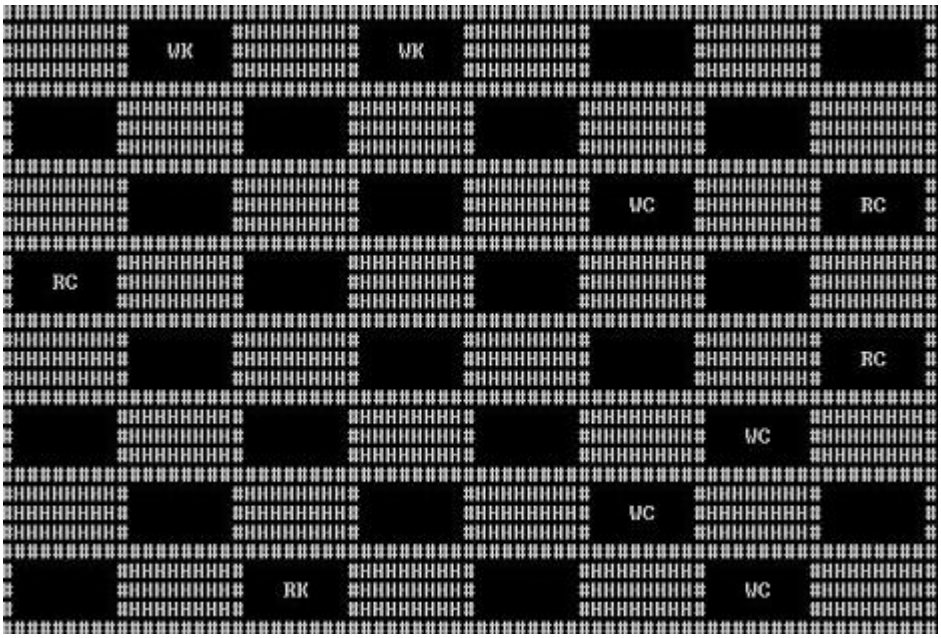

Figure 3.1: Board Position Showing the Initial Position in an Endgame Against a Human Opponent

The final board position of that game, where Soar Checkers allowed the human opponent to neutralize the significant material and positional advantage it had at the end of the middle game and escape with a draw, is recreated and shown in Figure 3.2 on the following page.

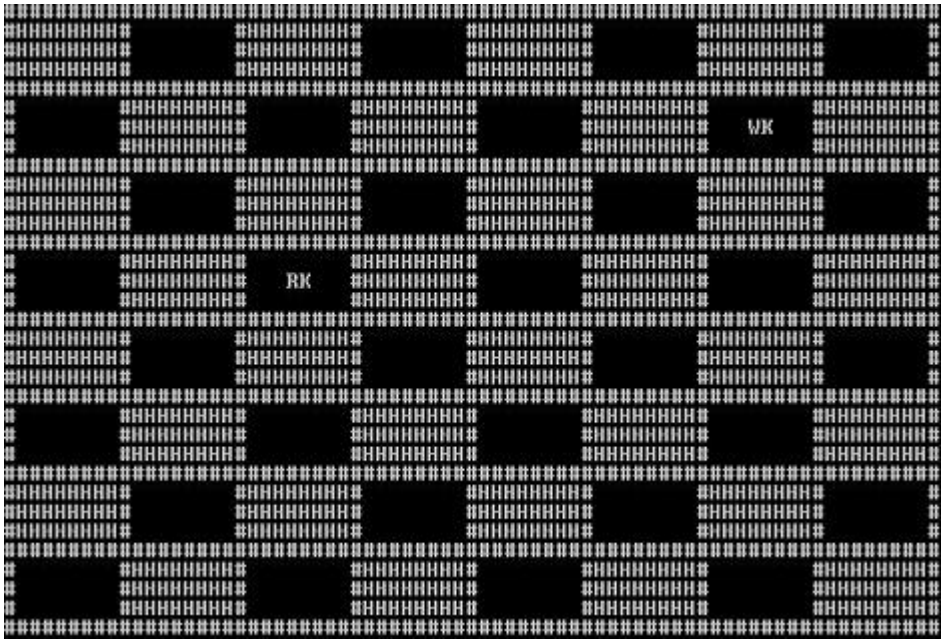

Figure 3.2: Board Position Showing the Final Position in an Endgame Against a Human Opponent

3.2.3 Soar versus Soar

One of the key ways in which checkers differs from chess, (other than in the number of valid squares on the board, the kinds of pieces, and the number of pieces) is that in chess, it is almost always advantageous to be able to move first, since this allows the player (the one playing with the lighter pieces, usually white) to choose an opening that he or she is very familiar with, and might exploit weaknesses in the opponent's preparations for the match. However, in checkers, the pieces can only move forward unless it is a king, and it is highly unlikely that a player would have a king in the opening phase of the game. This has the effect that sooner or later, one of the two opposing sides would run out of safe moves (that is, moves that do not result in uncompensated losses of pieces), and quite often, this misfortune falls on the player who moved first (the one playing with the dark pieces, usually red, but also referred to as black in the literature). The result is that it is more often than not

a disadvantage to play as Red, which automatically means that that player makes the first move.

To see if this is indeed the case, Soar Checkers was run in agent versus agent mode. Both agents were identical, except in the case of the color of their pieces - one was Red, the other White. A total of one hundred games were played, and the results were noted. It was found that the agent playing as White won sixty eight games, the agent playing as Red (black) won twenty six games, and six games were declared as drawn. It can be seen that there is a huge disparity in the number of games won, depending on which color the agent plays, and the disparity is indeed in favor of the player going second, the White agent. 68% of all games were won by the White agent, while if we consider only those games that generated an outright winner, the winning percentage creeps up further to 72.34%. **Thus, Soar Checkers seems to back up the generally accepted assertion among the checkers community that it is almost always an advantage to play as White, in checkers**.

3.3 Future Directions

3.3.1 Enhance Strategies Used

In this section, some ideas for extending the work done so far in creating Soar Checkers will be discussed. Considerable effort has been made to weed out strategies that proved to be not the most advantageous in real game play, and keep only the ones that consistently yielded good results. However, it is not beyond the realm of possibilities that there might still be some very good strategies that haven't yet been incorporated into Soar Checkers. The design of Soar Checkers makes it relatively straightforward to add new strategies to the agent's rule base, and immediately test it out. The best way to do this is to actually see how the modified agent performs in a game against the old agent. Ideally, the performance of the

modified agent should be evaluated over a series of games, with each agent alternating colors so that the inherent advantage that the agent that plays as White has, is neutralized and does not skew the results.

3.3.2 Compare with Pure Minimax

Another idea that looks interesting is to develop a checkers program that uses some of the standard techniques described in Chapter One, specifically, the minimax algorithm to aid in its game-play, and then compare it with Soar Checkers. Specifically, it would be interesting to see if and when the moves chosen by both the programs match, and if they do, at roughly what depth of search this occurs, and so on.

3.3.3 Add an Opening Book

It was mentioned earlier that the lack of an opening book seriously hurts the performance of Soar Checkers, especially against opponents (human or computer) that have learnt or stored opening books. It might be worth investigating how much of a difference it would make (there is absolutely no doubt that there would indeed, be a positive difference) in the quality of Soar Checkers' game-play if some sort of opening book were used to aid its decision making process during the opening phase of a game. The problem with implementing this idea, of course, is that assembling a decent-sized opening book is a potentially laborious process involving lots of research and manual typing of the opening moves that are chosen to be included in the book.

3.3.4 Investigate Potential Bug

While developing Soar Checkers, a potential bug in Soar was uncovered. This had specifically to do with the rules that were written to implement triple jumps by kings. Though the logic for this rule was exactly the same as that for a triple jump by a checker, with the only difference being that the jumps could be in either forward or backward direction in the case of the king, unlike in the case of an ordinary checker, which is restricted to jumping in only one direction, it was found that whenever the agent was required to apply this rule, it caused the program to crash, with virtual memory getting eaten up completely. A lot of effort was put in to try and understand what was causing this mysterious behavior, but in the end, it required a rewrite of the rule into eight different rules for the program to run, without crashing. The reason for this crash, and that too for just one specific rule, is still unknown (though most signs point to infinite recursion, it is not known yet exactly what is causing it) and would be an interesting problem to look into in the future.

3.3.5 Add an Endgame Database

Another area where Soar Checkers' game play could be improved substantially is in the endgame. As explained before, the kind of game-play required in endgames is impossible to describe using just strategies, as most of the time, precise moves must be made to precise squares in specific orders in order to convert an advantageous position into a victory, or to recover from an inferior position and achieve a draw. Though a lot of effort would be required to achieve this, it would be highly interesting to see how a potential future version of Soar Checkers that has been augmented with an endgame database would perform against human opponents, or even Chinook.

3.3.6 Conclusions

It is a fairly good guess that adding an opening book and an endgame database to the current version of Soar Checkers would go a really long way in leveling the playing field, especially against accomplished human opponents or Chinook, but in the end, relying on pure strategy may not be enough to win against highly efficient and deep brute force search. At the moment of writing, the Chinook team have announced that they have already solved (computed the game theoretic value of) seven different standard checkers openings. With the increased availability of cheap processing power, it is only a matter of time before their ultimate goal, which is to solve the game of checkers would be achieved.

This experiment does raise some interesting questions about Artificial Intelligence, and intelligence, in general. The current version of Soar Checkers relies almost entirely on strategy, and though it finds it relatively simple to beat human opponents who are either novices or advanced novices, it was beaten easily by the online version of Chinook, playing at novice level (though, as explained before, even at novice level, it uses the opening book and the full and perfect six-piece endgame database). It is a known fact that all players who play, or aspire to play, at the elite level spend considerable time and effort in memorizing thousands of opening lines, and even more endgame positions (apparently, the best players have perfect knowledge of any endgame involving five or less pieces of any kind!). Someone who is a novice, or just plays checkers for fun, on the other hand, is more likely than not going to rely on what they see on the board rather than memory. The question, therefore, is what should the goal of an artificially intelligent program be? Should it aim to be perfect (which seems to be the case with Chinook, and they're well on the way to achieving that)? Should it aim to emulate humans? It can be safely said that most humans are nowhere near perfect, and in this respect, Soar Checkers does seem to succeed in emulating average human

behavior, at least in the case of playing a game of checkers. On the flipside, Chinook has emulated, and even bettered the behavior of the Grandmasters (It should also be noted that its stated goal was to build a program capable of beating the word champion - which, it did).

APPENDICES

Appendix A

Brief History and Rules of Checkers

A.1 History

Checkers is one of the oldest known games played by man, with the earliest form of the game found at an archaeological site in Ur, Iraq, that was dated back to approximately 3000 B.C. [14] As far as 1400 B.C, a popular game called Alquerque was played in Egypt on a 5 x 5 board, which was later modified by a Frenchman so that it could be played on a standard chess board, thus increasing the number of pieces to twelve, per side. This version of the game was called "Fierges" or "Ferses", which was further modified to make jumps mandatory, yielding "Jeu Force" - which was exported to Britain and the Americas later. The game came to be known as "draughts" in Britain, and "checkers" in America. A brief summary of the rules of modern American checkers [15, 16] follows:

A.2 Rules

Checkers is a board game played between two players who alternate turns. The player who cannot move, either because he or she has no pieces left, or all of his or her pieces are blocked, loses the game. The players are also free to resign, or agree to a draw.

The game is played on a square board made up of 64 smaller squares arranged in an 8 x 8 grid, with the smaller squares alternately light and dark colored (green and buff, in the case of tournaments). The game is played on the dark squares (though in checkers diagrams, the pieces are shown on the light squares, for readability). The pieces are red and white (though more often than not, store-bought sets have black and white pieces), and at the beginning of the game, each player has twelve pieces of his or her chosen color arranged on the twelve

dark squares closest to his or her edge of the board. The red player moves first, and the players alternate turns until one of them can't make a move (in which case, he or she loses), or one of them resigns or they agree to a draw.

A piece that is not a king may move one square forward, diagonally. A king can move one square diagonally either forward, or backward. A piece can move only to a vacant square. A move may consist of one or more jumps. An opponent's piece can be captured by jumping over it, diagonally to the adjacent vacant square in the same diagonal. The jumping piece, the opponent's piece and the empty square must line up diagonally. While a king can jump diagonally, both forward and backward, a piece that is not a king can only jump diagonally forward. Using only one piece, a multiple jump can be made, moving from empty square to empty square. A jumping piece or king can change directions during a multiple jump - making a jump in one direction, and then jumping in another. Only one piece can be jumped with a given jump, but several pieces can be jumped with one move that involves multiple jumps. If a jump can be made, it must be made. However, if there are several different jumps that are possible, the player is free to choose among them, regardless of if some of them are multiple jumps, or not. A player cannot jump his or her own pieces. The same piece cannot be jumped twice, in the same move.

When a piece reaches the last row (usually known as the king row), it gets crowned, and becomes a king. This is indicated by placing a second checker on top of the first piece, by the opponent. A piece that has just been crowned cannot continue to jump opponent pieces until the next move.

A Draw is declared when neither player can force a win. When one side appears stronger than the other and the player with what appears to be the weaker side requests the Referee for a count on moves, then, if the Referee so decides, the stronger party is required to

complete the win or show to the satisfaction of the Referee at least an "increased" (instead of the old wording "decided") advantage over his or her opponent within forty of his or her own moves, these to be counted from the point at which notice was given by the Referee; failing in which he or she must relinquish the game as a draw.

[This forty-move draw rule requires some checkers expertise, on the part of the Referee. He or she must determine if a player has increased his or her advantage. There is no "triple-repetition" rule. So, a player may repeat the position, twenty times (or a hundred times, if the Referee has not been called in). Normally, the players are reasonable enough to agree to a draw, in such a case. Technically, a player who does not have any advantage, can refuse to agree to a draw. The player with the stronger position cannot request a 40 move count, at least not according to this law. So, it is possible to have a repetitious game that lasts forever. Reasonable people would agree to the draw, or apply the 40 move draw rule, anyway.]

Appendix B

Overview of Soar

B.1 Introduction

Soar is a general, unified cognitive architecture designed to develop systems exhibiting intelligent behavior. In other words, Soar provides the fixed computational structures that enable knowledge to be encoded and the production of actions in pursuit of goals. In many ways, it is similar to a programming language, albeit somewhat specialized. It differs from other programming languages such as C or Java in that it has embedded in it a specific theory of the appropriate primitives underlying reasoning, learning, planning and other capabilities that are necessary for intelligent behavior. Since Soar is not a general purpose language, some computations are awkward, or difficult to do in Soar (for example, complex math) but it is suited to build autonomous agents that use large bodies of knowledge to generate actions in pursuit of goals [17].

B.2 Overview

The design of Soar is based on the idea that all deliberate goal-oriented behavior can be viewed as the selection and application of operators to a state. A state is a representation of the current problem-solving situation; an operator transforms the state (makes changes to the representation) and a goal is the desired outcome of the problem-solving activity. Thus, as Soar runs, it is continually trying to apply the current operator and select the next operator (a state can have only one operator at a time) until the goal has been achieved. There are separate memories in Soar for short-term memory (for example, the current state of the problem) and long-term knowledge. A Soar program contains the knowledge for solving a

specific task (or set of tasks), including information about how to select and apply operators to transform the states of the problem, and means to recognize if and when the goal has been achieved.

B.2.1 Representation of States, Operators and Goals

States are represented as objects in Soar's short-term memory (known as "working memory" in Soar parlance), and these objects are made up of (a possible collection of) attribute-value pairs and in many cases, other objects are sub-structures of the state, forming a hierarchical representation. For example, the state representing a human could possibly have the following structure:

```
state   <human>   ^type      state
<human>       ^lastName      Smith
              ^firstName     John
              ^address       <placeHolder>
              <placeHolder>  ^Street '550~Old Broken Hwy'
                             ^City     'Utopia'
```

A state can have only one operator at a time, and operators are represented as substructures of the state. It should be noted that a state could have as a substructure a number of potential operators that are under consideration; however there still will only be one current operator. Meanwhile, goals are either represented explicitly as some substructure of the state with general rules that recognize when the goal is achieved, or are implicitly

represented in the Soar program by goal-specific rules that test the state for specific features and recognize when the goal is attained.

B.2.2 Problem Solving in Soar

As explained before, Soar tries to solve a problem by continuously applying the current operator (to the current state) and selecting the next operator, until the goal has been reached. The various steps involved are briefly described below.

B.2.2.1 Operator Proposal

As a first step in selecting an operator, one or more candidate operators are proposed. Operators are proposed by rules that test the features of the current state (for example, in a game of checkers, an operator to generate moves could be proposed when it is detected that it is that agent's turn to make a move).

B.2.2.2 Operator Comparison

The second step Soar takes in selecting an operator is to evaluate or compare the candidate operators. In Soar, this is done through rules that test the operators that were proposed during the operator proposal stage by using a mechanism known as "preferences" (for example, if there are two proposed operators A and B, each representing possible moves, if A leads to an uncompensated loss while B is safe, preference rules could be written that would prefer operator B over operator A).

B.2.2.3 Operator Selection

Soar attempts to select a single operator based on the preferences available for the candidate operators (in other words, operator selection is handled by the Soar architecture, not the Soar program). There are four possible scenarios here:

1. The available preferences unambiguously prefer a single operator.
2. The available preferences suggest multiple operators, and prefer a subset that can be selected from randomly.
3. The available preferences suggest multiple operators, but neither case 1 or 2 above hold.
4. The available preferences do not suggest any operators.

In the first case, the preferred operator is selected. In the second case, one of the subset is selected randomly. In the third and fourth cases, Soar has reached something known as an "impasse" in problem solving, and a new substate is created. There are four kinds of impasses which are explained in great detail in [8]:

1. Tie-impasse
2. Conflict-impasse
3. Constraint-failure impasse
4. No-change impasse

B.2.2.4 Operator Application

An operator applies by making changes to the state; the specific changes that are appropriate depend on the selected operator and the current state. There are two primary

approaches to modifying the state: indirect and direct. Indirect changes are used in Soar programs that interact with an external environment: The Soar program sends motor commands to the external environment and monitors the external environment for changes. The changes are reflected in an updated state description, garnered from sensors (example, how the agents interact with the game environment in Soar Checkers). Soar may also make direct changes to the state; these correspond to Soar doing problem solving "in its head". Soar programs that do not interact with an external environment can make only direct changes to the state.

B.2.2.5 State Elaboration

Making monotonic inferences about the state is the other role that Soar long-term knowledge may fulfill. Such elaboration knowledge can simplify the encoding of operators because entailments of a set of core features of a state do not have to be explicitly included in application of the operator. In Soar, these inferences will be automatically retracted when the situation changes, such as through operator applications or changes in sensory data.

B.2.3 Memories in Soar

There are three different kinds of memories in the Soar architecture, each of which has its own utility in the overall scheme of things, and will be described briefly in the following sections.

B.2.3.1 Working Memory

Soar represents the current problem-solving situation in its working memory. Thus, working memory holds the current state and operator (as well as any substates and operators

generated because of impasses) and can be thought of as being Soar's "short-term" knowledge, reflecting the current knowledge of the world and the status in problem solving. The working memory is made up of elements known as working memory elements (or WMEs). Each WME is an identifier-attribute-value triple (for example, J1 ^name Jack is a WME in which J1 is the identifier, "name " is the attribute and "Jack" is its value), and a collection of WMEs sharing the same identifier is said to form an object. The value of the attribute of a WME could be the identifier for another object, thereby creating an object hierarchy. Each object has to be connected to a state (which itself is represented as an object, as explained earlier), and any object that is disconnected from a state will be removed from the working memory.

B.2.3.2 Production Memory

Soar represents long-term knowledge as productions that are stored in production memory. Each production has a set of conditions and a set of actions. If the conditions of a production match working memory, the production fires, and the actions are performed. Thus Soar productions are very similar to "if-then" conditions.

Examples of Productions - The general structure of a Soar production is:

```
sp {rule*name
        (condition)
        (condition)
             .
             .
```

```
-->
        (action)
        (action)
}
```

A simple Soar production that prints out 'Hello World!' and stops could look like this [17]:

```
sp {hello*world
   (state <s>     ^type state)
-->
   (write |Hello World!|)
   (halt)
}
```

Architectural Roles - Soar productions are the mechanism used by the Soar architecture to fulfill four roles used in problem solving:

1. Operator Proposal
2. Operator Comparison
3. Operator Application
4. State Elaboration

Persistence – The two main actions of a production are to create preferences for operator selection, and create or remove working memory elements. For operator proposal

and comparison, a production creates preferences for operator selection. These preferences should persist only as long as the production instantiation that created them continues to match. When the production instantiation no longer matches, the situation has changed, making the preference no longer relevant. Soar automatically removes the preferences in such cases. These preferences are said to have "I-support" (for "instantiation support"). Similarly state elaborations also have I-support.

On the other hand, WMEs created as a result of an operator application rule have what is known as "O-support" (for operator support). This is different from I-support in that the results of an operator application production (usually, adding or removing elements to or from the working memory) should persist even when that operator is no longer the current operator or that operator application production instantiation no longer matches. Working memory elements that participate in the application of operators are maintained throughout the existence of the state in which the operator is applied, unless explicitly removed (or if they become unlinked). Working memory elements are removed by a reject action of a operator-application rule.

B.2.3.3 Preference Memory

The selection of the current operator is determined by the preferences in preference memory. Preferences are suggestions or imperatives about the current operator, or information about how suggested operators compare to other operators. Preferences refer to operators by using the identifier of a working memory element that stands for that operator. After preferences have been created for a state, the decision procedures evaluate them to select the current operator for that state. For an operator to be selected, there will be at least one preference for it, specifically, a preference to say that the value is a candidate for the

operator attribute of a state (this is done with either an "acceptable" or "require" preference). There may also be others, for example to say that the value is "best". The different kinds of preferences and the symbols used to denote them in Soar are [8]:

1. Acceptable (+)
2. Reject (-)
3. Better (>), Worse (<)
4. Best (>)
5. Worst (<)
6. Indifferent (=)
7. Numeric - Indifferent (= number)
8. Require (!)
9. Prohibit (~)

B.2.4 Soar Execution Cycle

A Soar program executes through a number of cycles. Barring the case of impasses and the creation of substates, a Soar execution cycle consists of five phases:

1. Input: New sensory data comes into working memory. This is done through the ^io.input-link attribute of the agent.
2. Proposal: Productions fire (and retract) to interpret new data (state elaboration), propose operators for the current situation (operator proposal), and compare proposed operators (operator comparison). All of the actions of these productions are I-supported. All matched productions fire in parallel (and all retractions occur in

parallel), and matching and firing continues until there are no more additional complete matches or retractions of productions (known as "quiescence").

3. Decision: A new operator is selected by Soar, or an impasse is detected and a new state is created.
4. Application: Productions fire to apply the operator (operator application). The actions of these productions will be O-supported. Because of changes from operator application productions, other productions with I-supported actions may also match or retract. Just as during proposal, productions fire and retract in parallel until quiescence.
5. Output: Output commands are sent to the external environment. This is done through the agent's ^io.output-link attribute. The cycles continue through phases 1 to 5 until the halt action is issued from the Soar program (as the action of a production) or until Soar is interrupted by the user.

This concludes the fairly broad overview of Soar. For further details, the reader is advised to refer to [8, 9, 10, 17].

BIBLIOGRAPHY

[1] Jonathan Schaeffer et al. A World Championship Caliber Checkers Program. *Artificial Intelligence*, vol. 53 (2 – 3): 273 – 290, 1992.

[2] Francis Heylighen. Zero Sum Games. Retrieved June 11, 2006 from the World Wide Web: http://pespmc1.vub.ac.be/zesugam.html, 1993.

[3] S. J. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2nd edition, 2003.

[4] K. M. Chandy. Explanation of Minimax. Retrieved May 14, 2006 from the World Wide Web: http://www.cs.caltech.edu/~vhuang/cs20/c/research.archetype.html.

[5] Wikimedia Inc. Minimax. Retrieved August 10, 2006 from the World Wide Web: http://en.wikipedia.org/wiki/minimax_theorem, 2006.

[6] Wikimedia Inc. Alpha-Beta Pruning. Retrieved October 21, 2006 from the World Wide Web: http://en.wikipedia.org/wiki/alpha-beta_pruning, 2006.

[7] Jonathan Schaeffer. The History Heuristic and the Performance of Alpha-Beta Enhancements. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol 11 (11): 1203 – 1212, 1989.

[8] J. E. Laird and C. B. Congdon. *The Soar User's Manual Version 8.6*. Electrical Engineering and Computer Science Department, University of Michigan, 2006.

[9] Douglas Pearson. *SML Quick Start Guide*. Three Penny Software LLC, 2006.

[10] Douglas Pearson. *Guide for Moving from SGIO to SML*, 2005.

[11] Derek Oldbury. *Move Over or How to Win at Draughts*. Published by Nicholas Kaye, 1957.

[12] Michael J. Mefford. A Checkers Game for Your PC. PC Magazine, vol v14v9, May 1996.

[13] Department of Computer Science, University of Alberta. Chinook. Retrieved November 9, 2006 from the World Wide Web: http://www.cs.ualberta.ca/~chinook.

[14] W. J. Rayment. History of Checkers or Draughts. Retrieved November 5, 2006 from the World Wide Web: http://www.indepthinfo.com/checkers/history.shtml, 2006.

[15] Jim Loy. The Standard Laws of Checkers. Retrieved August 3, 2005 from the World Wide Web: http://www.jimloy.com/checkers/rules.htm, 1995.

[16] Jim Loy. The Basic Rules of Checkers. Retrieved August 3, 2005 from the World Wide Web: http://www.jimloy.com/checkers/rules2.htm, 1999.

[17] J. E. Laird. *The Soar 8 Tutorial*. Electrical Engineering and Computer Science Department, University of Michigan, 2006.