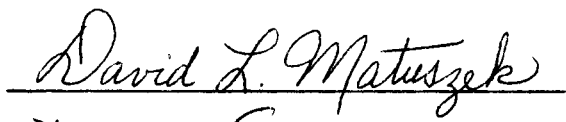
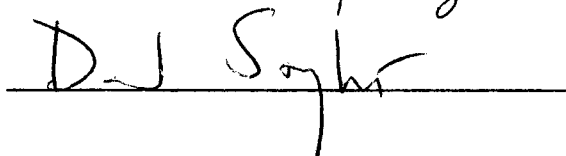


To the Graduate Council:

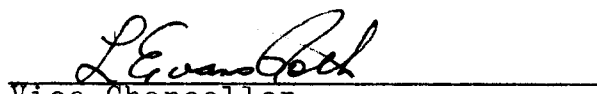
I am submitting herewith a thesis written by Robert Gerald Bryan entitled "A Solution to the Baker Game Problem." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.


Charles Pfleeger, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:


Vice Chancellor
Graduate Studies and Research

A SOLUTION TO THE BAKER
GAME PROBLEM

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Robert Gerald Bryan
December 1981

3055447

Copyrighted 1981

by

Robert Gerald Bryan

All rights reserved. No part of this thesis may be reproduced in any form without permission in writing from the author.

DEDICATION

This thesis is dedicated to Martin Gardner, whose columns in Scientific American sparked much of my early interest in mathematics, and whose column about the Baker game provided the subject for this thesis. It is especially appropriate that this particular thesis be dedicated to Dr. Gardner at this time, as he is retiring his column from Scientific American in December of this year.

ACKNOWLEDGMENTS

The author is grateful for the assistance of his thesis committee: Dr. Charles Pfleeger, Dr. David Straight, and Dr. David Matusek. Appreciation is also expressed to Mr. Craig Thompson, who provided most of the criticism and suggestions for improvement in the early drafts of the thesis.

The author acknowledges the use of the University of Tennessee Computing Center, Knoxville, Tennessee, in obtaining the results shown in this thesis.

Finally, the author wishes to thank his wife, Cathy Bryan, who encouraged him to resume work in school and who suffered through the many nights the author spent at the Computing Center, and without whom the thesis would not have been completed (but who, nevertheless, has informed the author that if he ever does this again, his second wife can encourage him the next time around).

ABSTRACT

The purpose of this thesis was to solve the Baker game, a one-person, complete information card game. The thesis is a case study in the use of game theory and artificial intelligence techniques to solve a non-trivial game.

The Baker game was solved with a computer program which implements an exhaustive search of the Baker game search space. A large number of algorithms are used by the program to reduce the size of the search space. The algorithms are proven to produce reduced search spaces which are equivalent to the original search space.

The program was able to solve about 90% of a random sample of different Baker game configurations. About 75% of initial Baker game configurations were shown to be winnable, and about 15% of initial Baker game configurations were shown to be unwinnable. The other 10% of initial Baker game configurations exceeded the capabilities of the program. These results represent significant improvements over human play of the Baker game, whereas many artificial intelligence studies produce a level of play inferior to human game playing.

The strategies used to reduce the size of the Baker game search space were specific to the Baker game, but some generalization to other games should be possible. The

strategies would be most adaptable to other one-person games.

TABLE OF CONTENTS

SECTION	PAGE
I. Artificial Intelligence, Game Theory, and the Baker Game	1
Artificial Intelligence	1
Game Playing and Game Theory	1
Study of the Baker Game	2
Terminology from Game Theory	3
Evaluation Functions in Artificial Intelligence	5
Limiting Searches in Artificial Intelligence	7
The Baker Game	9
The Baker Game Problem	10
Equivalence of Search Spaces	11
Baker Game Rules	13
Baker Game Terminology	15
The Baker Game Search Space	16
Sampling Criteria	18
Practicality of a Machine Solution	19
Necessity for Strategies	21
Heuristics	22
II. Baker Game Strategies	26
Introduction	26
Neighborhood Concept in Strategies	27
T-move Reduction	28
Dead end Anticipation	31

SECTION	PAGE
Efficacy of T-move Reduction	31
P move Preference	32
Efficacy of P move Preference	36
Proof of P-move Preference	36
P-move/T-move Combinations	37
B move Preference	39
Efficacy of B-move Preference	40
B move/T move Combinations	44
Empty B columns	45
Reversible Moves	48
Successive Move Suppression	51
B move Suppression	52
E-move Suppression	53
T-move Suppression	53
B*-move Suppression	54
E*-move Suppression	55
T*-move Suppression	55
III. The Baker Game Program	62
Programming Language	62
Program Management	62
Data Structures	63
Data Elements	65
Program Routines	70
Program Options	72

SECTION	PAGE
General Program Operation	74
The PLAY Routine	75
SEE Routine	77
Debugging and Program Validation	77
Development of New Strategies	78
IV. Results and Future Considerations	80
Necessity for Heuristics	80
Operation of the Program with Heuristics	80
The E-Frozen/T-Frozen Heuristic	81
Lack of Progress Heuristic	81
Speculations on the Undecided Games	81
Future Considerations	84
LIST OF REFERENCES	87
VITA	89

CHAPTER I

ARTIFICIAL INTELLIGENCE, GAME THEORY, AND THE BAKER GAME

Artificial Intelligence

Artificial intelligence is an area of computer science applications that interacts with other disciplines such as psychology, information theory, game theory, linguistics, philosophy, and pattern recognition. The goal of artificial intelligence is to model natural intelligence with machines. Artificial intelligence researchers build computational models that tie together some of the results from these other disciplines, using techniques borrowed from these disciplines as adapted for use on a computer.

Game Playing and Game Theory

Game playing and game theory are areas of artificial intelligence that are frequently investigated. The investigation of game playing by machines is popular because there is a large body of game theory upon which to draw, and because game theory has strong mathematical underpinnings. Also, game playing is an area in which it is fairly easy to make direct comparisons of artificial to natural intelligence. For instance, the capabilities of chess programs can be measured and assigned numeric rankings using the same system of measurement that is used for human

players. In many cases, the study of games provides a useful model of social and economic systems, and study of any particular game may provide techniques that are useful in the study of a large spectrum of other games.

Study of the Baker Game

This thesis describes the use of techniques of artificial intelligence and game theory in an investigation of a game called the Baker game. The Baker game is a solitaire card game that was described by Martin Gardner in his "Mathematical Games" column in the June, 1968 issue of Scientific American magazine.

The primary objectives in the investigation of the Baker game are two-fold. The first objective is to perform an analysis of the game. That is, the game will be studied using standard techniques from artificial intelligence and game theory to determine strategies which may be used to play the game well, and to determine the frequency distribution of winning and losing the game when playing well. In the case of the Baker game, the analysis by artificial intelligence proves to be better than that obtainable by natural intelligence alone. This aspect of the investigation may be regarded as a case in which artificial intelligence techniques are simply used to solve a particular problem.

The second objective is the development of new artificial intelligence techniques. Standard game theory search techniques are employed, but the standard techniques are modified to reduce the search time required to play the game.

Terminology from Game Theory

The simplest game theory technique for machine game playing consists simply of playing all possible sequences of moves for a game, and then choosing from these sequences of moves one sequence that best satisfies some criterion of winning the game or of playing the game well. The set of all positions or states for the game is called a search space. The search space is modeled by a search tree in which each node represents a state of the game, the root of the tree represents the initial state or position of the game, and the branches of the tree represent moves. The successor nodes for any state are the set of all states that can be reached on the next move. The terminal nodes represent states for which the game is completed. A completed game is usually considered to be won, lost, or tied, but for some games intermediate results between winning and losing are possible.

The branching factor below any node is the number of successors for the node. For a constant branching factor B ,

the number of paths through L levels is B^*L . The time required to search a tree is therefore critically dependent upon the value of the average branching factor B' . The closer B' is to 1 the smaller the tree, and even so modest a value for B' of 2 causes tree sizes to expand alarmingly as L increases.

The Baker game search space is searched depth first. For an example of a depth first search, consider the tree structure in Figure 1.

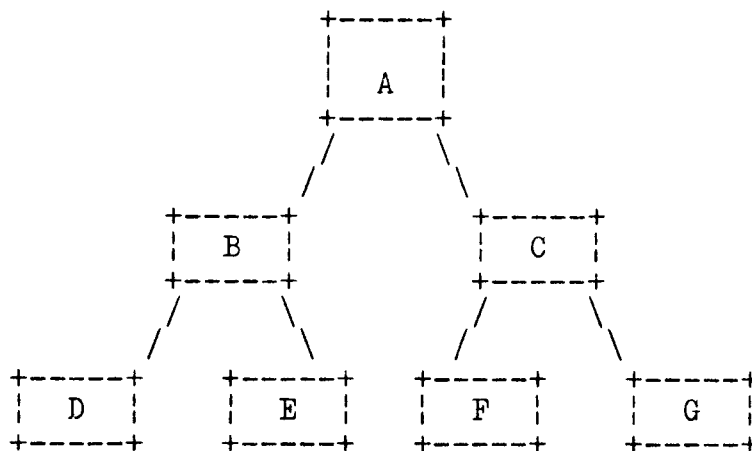


Figure 1

In Figure 1, a depth first search examines node A, node B, and node D. It then returns to node B and examines node E. After examining node E, the depth first procedure backtracks, returning to node B and node A, and then examines node C and node F. Finally, it returns to node C

and examines node G. The sequences of nodes ABD, ABE, ACF, and ACG are the paths through the tree.

Evaluation Functions in Artificial Intelligence

Tree search is a necessary but not sufficient component of game playing by machines. The exponential growth of search trees means that the search space for most games is too large to search completely, even with the fastest computer. Therefore, strategies are conventionally developed to reduce the search time.

Strategies typically involve an evaluation function that makes an estimate of the relative merits of various moves. An evaluation function is specific to a particular game, and is based on knowledge or experience concerning which moves tend to be good moves. In chess, for instance, moves that capture opposing pieces are often rated as good moves. The domain of an evaluation function is the state space, or set of all possible states, for the game. The range of an evaluation function is usually an interval such as 0 through 1 or -10 through +10. A state in which the game is won is usually assigned the largest value in the interval, and a state in which the game is lost is assigned the smallest value in the interval. Intermediate states are assigned appropriate intermediate values in the interval,

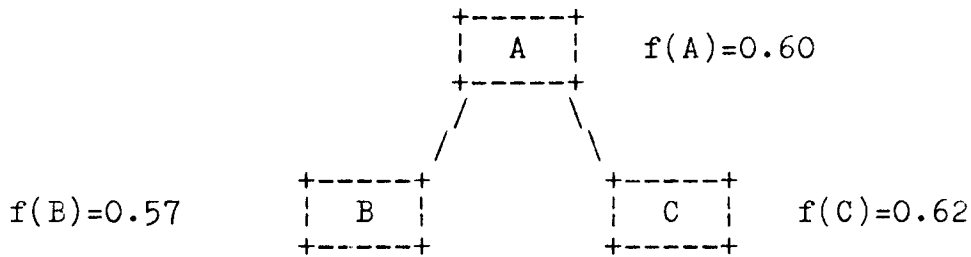
with the value indicating how close the game is to being won or lost.

If a perfect evaluation function for a given game existed, the function would represent a complete solution for the game because it would always permit the best possible move (or more correctly, a best possible move) to be made at each turn. In practice, perfect evaluation functions are not known for non-trivial games, and so evaluation functions are combined with a search procedure. The evaluation function serves as a guide to the search procedure, estimating the most fruitful paths to search. Conversely, the search procedure serves to refine and correct what is inherently an imperfect evaluation function.

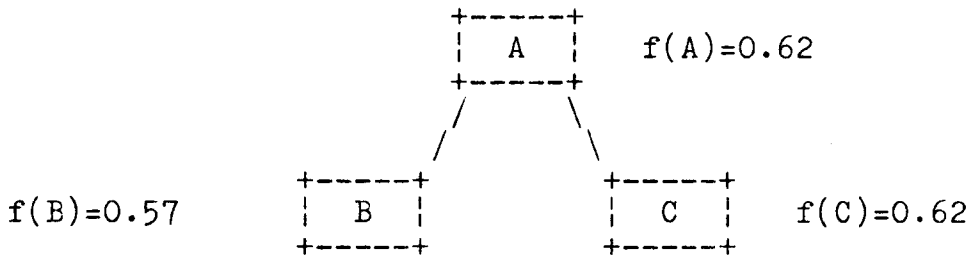
As an example of one interaction between a search procedure and an evaluation function, consider a node A for which the evaluation function returns a value of 0.60.

+-----+	
A	$f(A)=0.60$
+-----+	

Node A may have successor nodes B and C for which the evaluation function returns the values 0.57 and 0.62, respectively.



The value of node A may then be refined to be 0.62 because that is the maximum value of the function for the successors of A, and node C may be chosen as the best successor to node A.



The values of B and C may be similarly refined by examining their successors, and so forth for as many levels of the search tree as time permits. Such refinements are always propagated, or backed up, to predecessor nodes. For instance, a refined value for nodes B and C might be 0.59 and 0.58, respectively. The refined values for B and C would further refine the value for node A to be 0.59, and would indicate that node B rather than node C is a part of the most fruitful path to search.

Limiting Searches in Artificial Intelligence

Another interaction between search procedures and

evaluation functions involves limiting the scope of the search. The successors of a node may be ordered left to right based on the initial unrefined evaluation function, with the paths below the more promising nodes searched first. Also, the search of the paths below the less promising nodes may be eliminated entirely in a process called pruning to reduce the average branching factor for the tree and thereby to reduce the search space size and search time. Such pruning runs the risk that the best paths might be eliminated from the search space due to imperfections in the evaluation function.

In addition to ordering and pruning, using a search procedure in conjunction with an evaluation function always requires limiting the depth of the search in some fashion to keep the search time within reason. A search procedure with no limitations on search depth requires no evaluation function because an unlimited search needs only to recognize terminal nodes (nodes from which no further moves are possible), and to recognize whether a terminal node represents a state which is won or lost.

The search procedure used for the the analysis of the Baker game is an exhaustive procedure that does not use an explicit evaluation function. Nevertheless the Baker game uses strategies for reducing the search space, and moves below a given node are ordered according to which moves are

most likely to result in winning the game. Also, as will be seen in Chapter II, the notion of an evaluation function serves as a useful paradigm for establishing the validity of these strategies.

The Baker Game

The Baker game is a complete information one-person card game. By complete information, it is meant that all cards are shuffled and dealt face up. After the cards are dealt, the location of all of them is known so that chance and probabilities are not a part of the game after the cards are dealt.

The game is played with a standard 52 card deck. The ranks of the cards in each suit in a lowest to highest order for the Baker game are: ace, deuce, three, ..., ten, jack, queen, and king. The cards are manipulated according to the Baker game rules, leading eventually either to winning or losing the game. In the Baker game, there are no intermediate results between winning and losing, nor are there any degrees of winning or losing such as winning by a large or small margin.

After the cards are dealt, it may or may not be possible to win the game. A winnable game may be won by skillful play and may be lost by poor play. An unwinnable game will always be lost regardless of the skill with which

it is played. There are $52!$ unique initial states for the Baker game. Of these states, some number $\underline{n}$ of them are winnable and $52! - \underline{n}$ are unwinnable. The probability that a randomly chosen initial state is winnable is therefore simply $\underline{n}/(52!)$.

The Baker Game Problem

The only published literature concerning the Baker game is Gardner's article from Scientific American. Gardner's article was basically just an exposition of the rules of the game, with little discussion of strategy. The article did point out the difficulty of playing the game skillfully. It also made an intriguing and unsubstantiated remark to the effect that the probability of winning a randomly chosen initial state was "high" provided the player was skillful.

The author of this thesis played the game several times out of curiosity, and won the game a fairly low percentage of the time. The question naturally arose as to whether the poor results were the result of poor play or the result of the actual percentage of winnable games being low. The additional question also arose as to precisely what is the probability that a randomly chosen initial Baker game state is winnable. In other words, what is the value of $\underline{n}/(52!)$?

The answer to the question could be approximated by statistical sampling techniques provided it were possible to

determine precisely whether each initial state in a randomly chosen sample were winnable or unwinnable. Such a precise determination for each initial state could be attempted in one of two basic ways. On the one hand, each initial state could be analyzed in terms of the initial positions of the cards (i.e., by an evaluation function). The game proves to be too complex for a tractable perfect evaluation function to exist. On the other hand, each initial state could be analyzed by a search procedure which examined all sequences of moves originating at the initial state. Such a search can be performed in principle, but the number of sequences of moves originating at a given initial state for the Baker game is too large to search in practice.

Equivalence of Search Spaces

The question then still remains, "What is the value of $n/52!$?" The approach that is taken in this thesis is to take a random sample of states, and to categorize each initial state as winnable or unwinnable with a search procedure. Strategies are developed to search only a subset of the entire search space, and the subset selected by the strategies is shown to be equivalent to the entire search space. Equivalence is defined in terms of the results of the search. The subset is equivalent to the entire search space if and only if the results of searching both spaces is

exactly the same. Both searches must determine that the initial state is winnable or both searches must determine that the initial state is unwinnable.

Winning a Baker game consists of moving the cards into one particular final configuration, or goal state. A search space for an initial state which is winnable contains the goal state, and a search space for an initial state which is unwinnable does not contain the goal state. Thus, equivalence of a search space and any subspace is determined by whether or not the search space and the subspace in question contain the goal state. A search space for an initial state which is unwinnable does not contain the goal state, no subspace of such a search space can contain the goal state, and therefore a search space which does not contain the goal state is equivalent to any of its subspaces. A search space for an initial state which is winnable does contain the goal state, so a subspace of such a search space is winnable if and only if it also contains the goal state. In general, it is not known in advance whether a search space or a subspace is winnable or not. When it is not known whether an initial state is winnable or unwinnable, equivalence of the search space and the subspace may be defined in terms of the complement of the subspace; that is, equivalence may be defined in terms of the states that were eliminated from the search space to create the

subspace. A search space is equivalent to a subspace either if the complement of the subspace does not contain the goal state, or if the presence of the goal state in the complement of the subspace implies the presence of the goal state in the subspace itself.

Search procedures for games such as chess, checkers, and Go use strategies to reduce the size of the search space to a reasonable size, but the strategies are inevitably imperfect, and result in less than optimum play. The unique aspect of the Baker game program is that it seeks to reduce the search space for a non-trivial game in such a way that the chosen subspace is always equivalent to the original space. In this way, the game can be played perfectly and valid statistical inferences about the game may be obtained.

Baker Game Rules

To play the Baker game, the cards are dealt face up into an eight column array. The dealing is left-to-right, with seven cards in the first four columns, and six cards in the remaining four columns. The cards overlap so that only one card in each column is completely uncovered, but all the cards are visible. The columns will be called B-columns in this thesis (B1 through B8). Above the B-columns are four cells called P-cells (P1 through P4). The P-cells are initially empty and are intended to contain the cards of the

four suits in standard solitaire fashion. Below the B-columns are four cells called T-cells (T1 through T4). The T-cells are initially empty and are intended to be temporary cells to contain a single card in the course of the game.

The Baker game may also be played with two or three suits. With two suits, there are two P-cells, six B-columns, and two T-cells. With three suits, there are three P-cells, seven B-columns, and three T-cells. The two and three suit games are much easier to play than is the four suit game. The rules of the game are as follows:

1. Only one card may be moved at a time.
2. Only the top (uncovered) card may be moved from any B-column.
3. An ace may be played on any empty P-cell. On top of it may be played the deuce of the same suit, then the three, and so on through the king. These cards may come from the exposed ends of the B-columns or from the T-cells.
4. An uncovered card in a B-column or a card in a T-cell may be moved to become the uncovered card of a B-column only if it goes into an empty column or if it goes on the next higher card of the same suit.
5. An uncovered card in a B-column may be played on a T-cell and remain there until it is desired to move it somewhere else.
6. The game is won when all the cards are in the P-cells.

Baker Game Terminology

The three basic types of moves in the Baker game are P-moves, B-moves, and T-moves, where the type of move is determined by the type of cell to which the card is moved. From a given state S , these three types of moves are denoted as $P(S)$, $B(S)$, and $T(S)$, respectively. In addition, it proves useful in some cases to categorize B-moves further as I-moves, E-moves, and ordinary B-moves. I-moves occur when a card is moved from a T-cell to an empty B-column. E-moves occur when a card is moved from the exposed end of a B-column to another B-column which is empty. Ordinary B-moves occur when a card is moved to a B-column which is not empty.

Two types of indexing may be used to describe these moves. Moves may be indexed by card, or alternatively by location. When moves are described with indexing by card, the moves will be denoted as $P_c(S)$, $B_c(S)$, $E_c(S)$, $I_c(S)$, and $T_c(S)$, where the index c indicates the card which is moved. For instance, $P(K-D)(S)$ indicates that the K-D is moved to a P-cell. All empty B-columns are equivalent to each other. Also, all empty T-cells are equivalent to each other. Therefore, the Baker game program always moves cards to the left-most empty B-column when making I-moves and E-moves, and always moves cards to the left-most empty T-cell when making T-moves. The convention of using the left-most empty

B-column or T-cell makes a description of a move unique for all five types of moves when indexing by card.

The alternative description of a move indexes the location from that or to which the card is moved. For P-moves and ordinary B-moves, the index indicates the location to which the card is moved. $P_i(S)$ indicates that a card is moved to the i -th P-cell and $B_i(S)$ indicates that a card is moved to the i -th B-column. For instance, $P_4(S)$ indicates that the next card in the spade suit is moved to the fourth P-cell. For T-moves and E-moves, the index indicates the location from which the card is moved. $T_i(S)$ indicates that a card is moved to a T-cell from the i -th B-column and $E_i(S)$ indicates that a card is moved to an empty B-column from the i -th B-column. For instance, $T_8(S)$ indicates that a card is moved to a T-cell from the eighth B-column. I-moves are not indexed by location.

The Baker Game Search Space

The Baker search space is too large to search in its entirety, but it is useful to suppose it to have been searched and for definite but unknown values of an evaluation function to have been assigned to each node. Suppose each terminal node has either the value +1 for goal states or 0 for terminal states which are not the goal state. The terminal nodes with a 0 value for the evaluation

function are dead states (states from which no moves may be made). By the backing up procedure, each node in the entire search space will then have the value of +1 or 0. Nodes with a value of +1 are nodes from which the game may be won by proper play. Nodes with a value of 0 are nodes from which it is impossible to win the game. Within this context, it will be useful to state that for two states S_1 and S_2 , $f(S_1)=f(S_2)$. Such statements may often be made without knowing whether it is the case that $f(S_1)=f(S_2)=0$ or $f(S_1)=f(S_2)=1$, but the fact that $f(S_1)=f(S_2)$ alone is sufficient to make decisions about the search strategy, even though the exact value of the function is not known.

The Baker game typically might last 100 moves or so, with about 10 possible moves available below each node in the search tree. This should result in a game tree containing 10^{100} paths, an impossibly large number. The question naturally arises, how can an exhaustive search procedure possibly work with so large a search space? In addition, good statistical sampling practices would indicate selecting at least several hundred games for analysis. As a result, an informal design criterion of one minute of computer time per game was selected.

This objective must be interpreted rather loosely because of differences between software, hardware, and the speed of various machines. Extreme examples will be

presented which may clearly be categorized within the design objectives. In any case, the strategies employed by the program make it possible to meet the design objective for most randomly chosen initial states.

Sampling Criteria

A sample size of a few hundred games was chosen as a compromise between the high cost of analyzing each game on the one hand, and the improved reliability of estimates based on large sample sizes on the other hand. The reliability of the estimate is a function only of the sample size, and is independent of the size of the population. Within the constraint of a sample size of "several hundred", a 95% confidence interval for an error of 5% is a reasonable objective. The sample size for a 95% confidence interval with a 5% error is given by the formula $n = ((1.96^2) * p * q) / (0.05^2)$, where p is the proportion of successes in the sample, q is $1-p$, and n is the sample size. The worst possible case is $p=q=0.5$, for which $n=384$. The value of p is not known in advance. Indeed, discovering its value is one of the primary objectives of this thesis. Nevertheless, preliminary runs with extremely small samples suggested that p lay in the range of 70% to 80%. For values of p in this range, the sample size n is reduced to about

300. It was therefore decided to base the analysis of the Baker game upon a sample size of 300 games.

Practicality of a Machine Solution

There is certainly a question as to whether a machine solution is at all practical for the Baker game. The search space for each game has been shown to contain about 10^{100} paths, but the search time has been constrained to about one minute of computer time. There are two cases to consider in answering the question of practicality. The first case is when the initial state is winnable, and the second case is when the initial state is unwinnable. There are initial states which are extremely easy to categorize as winnable or unwinnable, and there are others that are extremely difficult. The answer to the question of practicality requires reference to an initial state for which it is trivial to find a winning solution, and reference to an initial state which is trivially demonstrated to be unwinnable.

It is easy to "stack the deck" to produce an unwinnable game. The following state is clearly unwinnable:

7-S	7-H	7-D	7-C	A-S	A-H	A-D	A-C
8-S	8-H	8-D	8-C	2-S	2-H	2-D	2-C
9-S	9-H	9-D	9-C	3-S	3-H	3-D	3-C
T-S	T-H	T-D	T-C	4-S	4-H	4-D	4-C
J-S	J-H	J-D	J-C	5-S	5-H	5-D	5-C
Q-S	Q-H	Q-D	Q-C	6-S	6-H	6-D	6-C
K-S	K-H	K-D	K-C				

The only moves possible in this game are T-moves. After four moves, the T-cells are full, no further moves are possible, and the game is lost

It is just as easy to "stack the deck" in the other direction. For instance, the following initial state clearly represents a winnable game:

K-S	Q-S	J-S	T-S	9-S	8-S	7-S	6-S
5-S	4-S	3-S	2-S	A-S	K-H	Q-H	J-H
T-H	9-H	8-H	7-H	6-H	5-H	4-H	3-H
2-H	A-H	K-D	Q-D	J-D	T-D	9-D	8-D
7-D	6-D	5-D	4-D	3-D	2-D	A-D	K-C
Q-C	J-C	T-C	9-C	8-C	7-C	6-C	5-C
4-C	3-C	2-C	A-C				

But what of a randomly chosen initial state? Such a state is either winnable or unwinnable. In order for exhaustive search categorizations to be possible, the game tree for a randomly chosen winnable or unwinnable initial state must be similar to a "stacked deck" that clearly represents a winnable or unwinnable state. If the similarities were extremely strong, the game and the program would be trivial, and therefore uninteresting. If the

similarities were extremely weak, the search tree would truly be a 10^{100} monstrosity, an exhaustive search would be impossible in practice, and the game and the program would therefore be uninteresting. The truth seems to lie between these two extremes. Similarities between randomly chosen games and "stacked deck" extremes are strong enough to make categorizations possible, but weak enough to make categorizations non-trivial.

Necessity for Strategies

This analysis should not be interpreted to mean that a Baker game program that simply knows the rules of the Baker game could be expected to solve the game, nor could a general problem solving program which was provided with the rules of the game be expected to solve the game. The analysis should only serve to indicate that it might be possible to solve the game with an exhaustive search despite the formidable size of the search tree.

A comprehensive set of search strategies is required to realize the goal of analyzing the Baker game with an exhaustive search procedure. These strategies are outlined in detail in Chapter II of this thesis, and estimates are given of the effect that they have on the size of the search space.

The desirable scenario for the program in playing a

winnable game is to make as many "good" moves as possible. That is, when a node has successor nodes, some of which are winning nodes and some of which are losing nodes, the program should choose first a winning node a large percentage of the time. In the cases where a losing node is chosen, the neighborhood of the losing node should degenerate as quickly as possible. That is, a losing node should have as few descendants as possible. The effect is that of traveling through the tree accompanied by a neighborhood of easily accessible nodes, generally making progress toward the goal, with occasional incorrect branches quickly discovered and eliminated.

Regardless of the efficacy of such strategies, it must be acknowledged that the seed for the success of the strategies lies in the structure of the game itself. The game has a fascinating and quaint property of always seeming to be on a bare edge, always nearly winnable and always nearly lost, without clearly being either. It is this tightrope of nearly winning and nearly losing that guides the program's strategies, and that makes solutions possible in the face of impossibly large numbers.

Heuristics

The goal in designing the Baker game program was to play the game in a fully admissible fashion. That is, the

goal was to prove rigorously that all strategies used to reduce the search space produced a sub-space equivalent to the original space. This effort was largely successful, but occasionally the fully admissible program failed to solve a game destined to be won, due primarily to problems toward the end of the game. The strategies developed to reduce the search space were fairly successful early in each game. Toward the end of a winnable game, there tended to be more possible moves below each node, and those extra moves tended to cause the search space to expand beyond the capabilities of the program. The irony with the end of game problems is that it is precisely in the end game that the human player finds it most easy to play the game. It was typical that the Baker game program might make the first 80 or 90 moves (the most difficult moves) successfully, yet not be able to complete the solution to the game. It was also typical that a human player could successfully (and quite easily) complete such a game by picking up at the point at which the program left off. As a result of difficulties with the end game, a number of heuristics were introduced into the program. That is, a number of procedures were introduced into the program to reduce the search space where the sub-space could not be proven to be equivalent to the original space. The heuristics are used only after a fully admissible procedure fails to decide the game. A

determination that a game is winnable using heuristics is totally valid. A determination that a game is unwinnable using heuristic procedures is tainted. Such a game may be winnable because heuristics do not guarantee admissibility. The safest conclusion with regard to a game found unwinnable via heuristics is that it is not known whether the game is winnable or not. A riskier conclusion is that the probability that such games are actually unwinnable is inversely related to the probability that a path to the goal state was eliminated by the heuristics. It is believed that the probability that the heuristics used by the Baker game program eliminate a path to the goal state is quite low, and that therefore the probability that games found unwinnable using heuristics are in fact unwinnable is quite high. Results of the runs of the program are thus summarized in Figure 2 both in terms of runs that are fully admissible, and runs that are heuristic. The Admissible column displays the results when no heuristics were used. Both the Winnable and Unwinnable figures are valid. The Heuristic column displays the results when heuristics were used. The Winnable column is valid, but the Unwinnable column cannot be considered to be valid. The Admissible and Heuristic column combines the best results when heuristics were and were not used. The Winnable figure is taken from the

Heuristic column, and the Unwinnable figure is taken from the Admissible column.

Four Suit Game					
Admissible		Heuristic		Heuristic and Admissible	
Winnable	-- 60%	Winnable	-- 73%	Winnable	-- 73%
Unwinnable	- 16%	Unwinnable	- 23%	Unwinnable	- 16%
Unknown	-- 24%	Unknown	-- 4%	Unknown	- 11%

Three Suit Game					
Admissible		Heuristic		Heuristic and Admissible	
Winnable	-- 84%	Winnable	-- 84%	Winnable	-- 84%
Unwinnable	- 14%	Unwinnable	- 16%	Unwinnable	- 14%
Unknown	-- 2%	Unknown	-- 0%	Unknown	- 2%

Two Suit Game					
Admissible		Heuristic		Heuristic and Admissible	
Winnable	-- 98%	Winnable	-- 98%	Winnable	-- 98%
Unwinnable	- 2%	Unwinnable	- 2%	Unwinnable	- 2%
Unknown	-- 0%	Unknown	-- 0%	Unknown	- 0%

Figure 2.

CHAPTER II

BAKER GAME STRATEGIES

Introduction

In general concept, most search strategies limit either the depth or breadth of the search (or both) to reduce the number of nodes which must be examined. The strategies in the Baker game program do not explicitly limit either the depth or the breadth of the search. Rather, the Baker game strategies eliminate certain sequences of moves. All the Baker game strategies take the following basic form. From state S , a sequence of moves $Q_1Q_2\dots Q_n(S)$ is eliminated from the search space. It is then demonstrated that there is some other sequence of moves $Q_1'Q_2'\dots Q_m'(S)$ which is not eliminated from the search space and which is either superior to, or equivalent to $Q_1Q_2\dots Q_n(S)$. For many of the Baker game strategies, either the sequence $Q_1'Q_2'\dots Q_m'$ is a permutation of $Q_1Q_2\dots Q_n$, or $Q_1'Q_2'\dots Q_m'$ is the sequence $Q_1Q_2\dots Q_n$ with useless moves omitted. Also, elimination of permutations and elimination of useless moves may be combined. In either of these two cases, $Q_1Q_2\dots Q_n(S) = Q_1'Q_2'\dots Q_m'(S)$. Justification of the decision to eliminate $Q_1Q_2\dots Q_n(S)$ requires, however, only that $f(Q_1Q_2\dots Q_n(S)) \leq f(Q_1'Q_2'\dots Q_m'(S))$.

Neighborhood Concept in Strategies

A useful concept in dealing with Baker game strategies is that of a neighborhood. The concept is exemplified in the "stacked deck" unwinnable initial state in Chapter I. In this, a worst possible initial state, it is nevertheless possible to advance four levels into the tree because four T-cells are empty. After the first move fills one T-cell, there is a guaranteed minimum of three additional moves. After two T-cells are full, there is a guaranteed minimum of two additional moves, etc. The number of empty T-cells therefore represents a neighborhood of nodes easily accessible with T-moves alone. The subtree below a node may be greater than the neighborhood due to moves other than T-moves, but the neighborhood defines a minimum subtree. There are 8 T-moves available below a given node, so the minimum size of a neighborhood below a node is 8^n paths, where n is the number of empty T-cells in the node.

As an example of the utility of neighborhoods to game analysis, suppose a kibitzer walks up to a person who is somewhere in the middle of a Baker game with three T-cells currently empty. The kibitzer can immediately state without further analysis that the player is in "pretty good shape" because he is guaranteed at least three more moves and there are at least 8^3 (or 512) ways of making those moves.

A strategy that both human players and the Baker game

program use is to keep the neighborhood as large as possible. That is, T-moves are avoided whenever possible, and moves which empty a T-cell are made whenever possible.

T-move Reduction

In an initial Baker game state there is a neighborhood containing 8×4 (or 4096) paths. It is not difficult to see that T-moves are commutative and that therefore many of these paths are equivalent. The Baker game program includes a strategy called T-move Reduction which causes commutative T-moves to be made in only one order.

For instance, suppose the initial state is S, and that the first move chosen consists of moving the king of clubs from the eighth B-column to a T-cell. This would be denoted as $T_8(S)$. Then suppose the second move consisted of moving the 9 of diamonds from the third B-column to a T-cell. This would be denoted as $T_3(T_8(S))$, or as $T_8T_3(S)$. It is clear that the two moves could be made in the other order with the same resultant state. That is, $T_3T_8(S) = T_8T_3(S)$, or more generally $T_iT_j(S) = T_jT_i(S)$.

In the special case of $i=j$, $T_iT_j(S) = T_jT_i(S)$ is a tautology, but a sequence $T_iT_i(S)$ should be interpreted carefully. When $T_iT_j(S) = T_jT_i(S)$ for $i \neq j$, the interpretation is clear; the two moves can be made in either order. On the other hand, the two moves $T_iT_i(S)$ should perhaps be

written as $TiTi'(S)$, because the moves are not interchangeable. When two cards are moved from the same B-column to the T-cells, the second card cannot be moved until it is uncovered by moving the first card. More generally, $TiTjTkTm(S)=Ti'Tj'Tk'Tm'(S)$, where $i'j'k'm'$ is any permutation of $ijkm$. The conditions $i \neq j$, $i \neq k$, $i \neq m$, $j \neq k$, $j \neq m$, and $k \neq m$, are not required. For instance, $TiTiTj=TiTjTi$. Although the two Ti moves are not commutative relative to each other, they are commutative relative to any Tj for $j \neq i$.

In order to discuss permutations of T-moves in the most general case, it must be noted that certain T-moves may be impossible. If one or more of the B-columns are empty, then it is impossible to make the corresponding T-move. For instance, if the fifth B-column is empty, it is impossible to make move T_5 because move T_5 removes a card from the fifth B-column and places it into a T-cell. The previous result that $TiTj(S)=TjTi(S)$ clearly also holds in the general case in which one or more B-columns are empty provided neither the i -th and the j -th B-columns are empty. When calculating neighborhood sizes, the size of a neighborhood below a state containing no empty T-cells should be interpreted as the upper limit for a state which does contain empty T-cells.

The calculation of neighborhood sizes when permutations

of T-moves are considered to be equivalent is as follows. When no T-cells are empty, the size of the neighborhood is zero in any case. When one T-cell is empty, the maximum size of the neighborhood is 8 paths in any case. When two T-cells are empty, the maximum size of the neighborhood is reduced from 64 paths to 36 paths when permutations are considered to be equivalent. When three T-cells are empty, the maximum size of the neighborhood is reduced from 512 paths to 120 paths when permutations are considered to be equivalent. When four T-cells are empty, the maximum size of the neighborhood is reduced from 4096 paths to 330 paths when permutations are considered to be equivalent.

The number of paths shown above is calculated from the formula for combinations of n things taken r at a time, where the r things need not be distinct. This formula is $(n+r-1)!/((n-1)!r!)$. For one empty T-cell, $8!/7!1!=8$. For two empty T-cells, $9!/7!2!=36$. For three empty T-cells, $10!/7!3!=120$. For four empty T-cells, $11!/7!4!=330$.

Given that $T_iT_j(S)=T_jT_i(S)$, it is possible to search either T_iT_j or T_jT_i and to eliminate the other sequence of moves from the search tree. The Baker game program makes the arbitrary choice to search T_iT_j where $i \leq j$ and to eliminate T_iT_j where $i > j$. More generally, the program searches $T_iT_jT_kT_m$ where $i \leq j \leq k \leq m$ and eliminates all other sequences of T-moves.

Dead-end Anticipation

One final reduction in neighborhood size is called dead-end anticipation. Dead-end anticipation detects a particular configuration which appears frequently enough in a Baker game tree to warrant special attention. This configuration consists of a state with exactly one empty T-cell when no moves except the eight T-moves are available. Any such T-move will result in an immediate dead-end unless it makes another move available. Usually, none of the eight T-moves does make another move available. It turns out to be fairly easy in a program to identify states which will result in a dead-end after one more move. The identification process is much faster than trying each of the eight T-moves, and results in effective reduction of the size of the neighborhood by one level in the game tree. The neighborhood size with four empty T-cells can therefore usually be reduced from 4096 paths to 120 paths (with a maximum of 330 paths); with three empty T-cells the neighborhood can usually be reduced from 512 paths to 36 paths (with a maximum of 120 paths), etc.

Efficacy of T-move Reduction

It is possible to estimate the effect of T-move reduction on tree size. In calculating tree sizes, it has been assumed that each node has approximately 10 successors,

of which 8 are T-moves and 2 are other moves. For the purpose of calculating the number of paths through a tree, the number of T-cells available below a node may be considered to be reduced from 8 to about 4 on the average by T-move reduction. This is because in choosing T_j to be applied to $T_i(S)$, j must be chosen such that $j \geq i$ and i varies from 1 through 8 in an approximately uniform fashion. Also, in some states no T-moves at all may be made. The total number of moves below each node is then about 6, and a typical tree for an entire game is reduced from approximately 10^{100} paths to approximately 6^{100} paths.

P-move Preference

It is not often the case that an initial Baker game state leads only to T-moves. It is common either in the initial state itself, or after one or two T-moves, that one or more P-moves may be made. Consider an initial Baker state in which one P-move is available. Such a state contains 9 successors, the P-move plus 8 T-moves. The addition of the P-move appears to increase the neighborhood of the initial state to at least five levels with at least 8 and sometimes 9 moves available at each level. This is because the P-move will remain available until it is actually made at each of the 5 levels. The size of the

neighborhood is thereby increased from 330 to approximately $12!/7!5!$, or 792 paths.

A human player typically treats a P-move as a "good" move, and will usually make it in preference to all others. This human strategy, called P-move preference, can be applied in the Baker game program. The presence of P-moves can be shown to add nothing to the effective size of a neighborhood, and to add essentially nothing to the search time of the program.

If an initial Baker state $\underline{S}$ has one P-move P_i available, then the 9 successors of $\underline{S}$ are $P_i(S)$, $T_1(S)$, $T_2(S)$, ... $T_8(S)$. For instance, if $i=3$ then the ace of hearts is uncovered and $P_3(S)$ plays the ace to a P-cell. If P_i is played as the first move, then it may be followed by any of the T-moves. The first two moves may then be $P_iT_1(S)$, $P_iT_2(S)$, ... $P_iT_8(S)$. However, the first move need not be P_i ; it could be some move T_j . If so, P_i is still available as the second move so that $T_jP_i(S)$ is certain to be a possibility as the first two moves.

The assertion that P_i is still available on the second move requires justification. Suppose two moves Q_a and Q_b are available at a particular node $\underline{S}$. Suppose move Q_b is chosen, so that $Q_b(S)$ is the successor of $\underline{S}$. Then, Q_a in general may or may not be available as the next move; that is, $Q_bQ_a(S)$ may or may not be a legal sequence of moves,

because making move Q_b may make move Q_a impossible. However, if Q and P_i are available at a given node $\underline{S}$, then it is easily demonstrated that $Q P_i(S)$ is always a legal sequence. That is, after a move P_i becomes available, it can become unavailable only by being made. In the case P_i moves an ace, the move cannot become unavailable because an ace cannot be covered up. In the case P_i moves a card $\underline{c}$ higher than an ace, the move cannot become unavailable because the card which could in general cover $\underline{c}$ must already be in a P-cell in order for P_i to be available.

Returning to the main argument, suppose $\underline{S}$ is any node with $P_i(S)$ and $T_j(S)$ as successors and consider $T_j P_i(S)$. Then, it is easy to show that either $T_j P_i(S) = P_i T_j(S)$ or $T_j P_i(S) = P_i(S)$. The result $T_j P_i(S) = P_i T_j(S)$ clearly holds when the cards moved by T_j and P_i are different because the two moves are simply permuted and they cannot interfere with each other. In this case, either sequence can be chosen, eliminating the other from the game tree and it is more convenient to retain $P_i T_j(S)$ and to eliminate $T_j P_i(S)$. The result $T_j P_i(S) = P_i(S)$ holds when the cards moved by T_j and P_i are the same card. In this case, T_j is a useless move and the sequence $T_j P_i$ can be eliminated from the game tree.

The preceding result with regard to P-moves generalizes nicely. If $\underline{S}$ is any node with successors $P_i(S)$ and $Q(S)$, where $\underline{Q}$ is any move other than P_i , then either $Q P_i(S) = P_i Q(S)$

or $Q\pi(S)=\pi(S)$. Therefore, $Q\pi(S)$ can be eliminated from consideration. Finally, if $\underline{S}$ is any node with successor $\pi(S)$ and descendant $Q_1Q_2\dots Q_n(S)$, where $Q_1Q_2\dots Q_n$ is any sequence of moves not including $\pi(S)$, then $Q_1Q_2\dots Q_n\pi(S)=\pi Q_1'Q_2'\dots Q_m'(S)$ where $0\leq m\leq n$ and $Q_1'Q_2'\dots Q_m'$ is the sequence $Q_1Q_2\dots Q_n$ with any move omitted which moves the same card moved by π . Note that the sequence $Q_1'Q_2'\dots Q_m'$ may be null in the event all moves Q_j move the same card as π . This result will be proven later by an induction proof. More simply, it means that any time a P-move π is available, it should be made immediately and other successors at the same level should be eliminated. This is true because if π is deferred, whenever it is finally made it will generate a state S' which is equal to a descendant of $\pi(S)$, and S' will therefore have already been examined in the search of the subtree below $\pi(S)$.

It may occasionally be the case that two (or more) P-moves π and π_j ($i\neq j$) are available at node S . If so, either one can be chosen arbitrarily and the other rejected. For instance, choose π if $i<j$, and all preceding results regarding π and Q remain valid when $Q=\pi_j$.

The consequence of the preceding results for P-moves is that for any node $\underline{S}$ including a successor $\pi(S)$, any other successors of $\underline{S}$ may be eliminated from the game tree. $\underline{S}$ then has only a single successor. The descendants of $\underline{S}$

which were eliminated are all still in the tree as descendants of $P_i(S)$. In terms of the evaluation function, $f(S)=f(P_i(S))$ whenever $P_i(S)$ is a successor of S .

Efficacy of P-move Preference

The effect of P-move preference on tree size is monumental. The number of paths through a tree is multiplicative at each level. P-move preference results in a multiplier of 1 at any level containing a P-move, leaving the number of paths through the tree unmodified by P-moves. Every solution of a Baker game includes exactly 52 P-moves. The initial $10^{**}100$ path tree, already reduced to approximately $6^{**}100$ by T-move reduction, is further reduced to approximately $6^{**}50$ paths by P-move preference.

Proof of P-move Preference

The following is a proof of P-move preference. In addition to establishing the validity of P-move preference, this proof is a model for a number of other similar proofs which are not included. The proof is a simple induction proof, and it is easy to see how other subsequent proofs could follow the same line of reasoning.

Theorem: If S is any node with descendants $P_i(S)$ and $Q_1Q_2...Q_n(S)$, where $Q_j \neq P_i$ for all j , then $Q_1Q_2...Q_nP_i(S) = P_iQ_1'Q_2'...Q_m'(S)$ where $0 < m < n$ and $Q_1'Q_2'...Q_m'$ is the sequence $Q_1Q_2...Q_n$ with any move omitted which moves the same card moved by P_i . Note that $Q_1'Q_2'...Q_m'$ may be null.

Proof: The proof is by induction. 1) The $n=1$ step is $Q_1 P_i(S) = P_i Q_1'(S)$ or $Q_1 P_i(S) = P_i(S)$, which has already been proven. 2) For convenience, let $T = Q_1 Q_2 \dots Q_n$ and $T' = Q_1' Q_2' \dots Q_m'$ where $0 < m < n$ and $Q_1' Q_2' \dots Q_m'$ is the sequence $Q_1 Q_2 \dots Q_n$ with any move omitted which moves the same card as P_i . Then the induction step in which the truth of the theorem for $n=k$ implies its truth for $n=k+1$ may be stated as follows: if $T P_i(S) = P_i T'(S)$, then $T Q(n+1) P_i(S) = P_i T' Q_z'(S)$, where $Q_z' = \phi$ if $Q(n+1)$ moves the same card as P_i , and $Q_z' = Q(m+1)' = Q(n+1)$ if $Q(n+1)$ moves a different card than P_i . Then, there are two cases:

a) $T Q(n+1) P_i(S)$ where $Q(n+1)$ moves the same card as P_i . By the induction hypothesis $T P_i(S) = P_i T'(S)$. Consider $T Q(n+1) P_i(S)$. Because $Q(n+1)$ moves the same card as P_i , $Q(n+1)$ is a useless move and $T Q(n+1) P_i(S) = T P_i(S)$. Therefore, $T Q(n+1) P_i(S) = P_i T'(S)$, and $Q_z' = \phi$.

b) $T Q(n+1) P_i(S)$ where $Q(n+1)$ moves a different card than P_i . By the induction hypothesis $T P_i(S) = P_i T'(S)$, so $T P_i Q(n+1)(S) = P_i T' Q(n+1)(S)$. Because P_i and $Q(n+1)$ move different cards, the two moves may be made in either order. Therefore, $T P_i Q(n+1)(S) = T Q(n+1) P_i(S)$ so that $T Q(n+1) P_i(S) = P_i T' Q(n+1)(S)$. Finally, let $Q(m+1)' = Q(n+1)$ with $(m+1) \leq (n+1)$, and $T Q(n+1) P_i(S) = P_i T' Q(m+1)'(S)$. QED

P-move/T-move Combinations

The previously identified symmetries with regard to T-moves have been predicated upon an unbroken sequence of T-moves; that is, $T_i T_j(S) = T_j T_i(S)$, but in general there is no guarantee that $T_i Q T_j(S) = T_j Q T_i(S)$ if Q is not a T-move. However, if $Q = P_k$, then the symmetries of T-move reduction and P-move preference may be combined to some extent.

For example, consider a Baker game state having cards as follows in the first two B-columns.

X-X	X-X
X-X	A-H
9-D	Q-S

In this state, T_1 moves the 9-D to a T-cell, T_2 moves the Q-S to a T-cell, and $T_1T_2(S)=T_2T_1(S)$ because the moves can be made in either order. But the sequence T_2T_1 would not be played in any case because of P-move preference. The move T_2 would always be followed by P_3 , moving the ace of hearts to a P-cell. The two move sequence T_2P_3 is in effect an inseparable entity, and it is clear that $T_1(T_2P_3)(S)=(T_2P_3)T_1(S)$, or that $T_1T_2P_3(S)=T_2P_3T_1(S)$. In other words, the same state results from moving the 9-D to a T-cell followed by moving the Q-S to a T-cell and the A-H to a P-cell as results from moving the Q-S to a T-cell and moving the A-H to a P-cell followed by moving the 9-D to a T-cell. In general, when the P-cell preference strategy is used, $T_jP_kT_i(S)=T_iT_jP_k$, where $i \neq j$. Note that the condition $i \neq j$ is mandatory; the case $i=j$ is not a tautology and in general $T_iP_kT_i(S) \neq T_iT_iP_k(S)$.

In both sequences, $T_iT_jP_k$ and $T_jP_kT_i$, P_k is guaranteed by P-move preference to move a card from the j -th B-column to a P-cell. The guarantee is because of the characteristic of P-move preference that a P-move is always made as soon as possible. Therefore, if move T_j was made in some state, move P_k must not be available in that state. Further, if move T_j is followed by move P_k , then move T_j must have made move P_k available. The only way for T_j to make P_k available is for move T_j to uncover the next card in the j -th

B-column, and for move Pk to move the uncovered card to a P-cell.

The combination of T-move and P-move symmetries represented by the equation $T_j P_k T_i(S) = T_i T_j P_k(S)$ for $i \neq j$ is implemented as a part of T-move reduction in the Baker game program. Normally, the program will follow move T_i by T_j only if $j \geq i$, but will follow a sequence $T_i Q$ for $Q \neq T$ with any move T_j . In the special case where $Q = P_k$, then $T_i P_k$ will be followed by T_j only if $j \geq i$.

B-move Preference

In addition to the 8 T-moves, there may be one or more P-moves in an initial Baker state. Similarly, there may be one or more B-moves in an initial Baker state, and B-moves may be made available at any lower level of the Baker game tree.

P-moves have been demonstrated always to be good moves, and the P-move preference strategy greatly reduces search time for the Baker game program. A similar, but weaker, result called B-move preference is available for B-moves. "B-move" here refers collectively to E-moves, I-moves, and ordinary B-moves. Practical experience playing the Baker game indicates that B-moves usually are good moves. If they exist, B-moves usually are the only moves which need be considered below a given node. Also, it may be proven that

if a B-move is a bad move at a given level in the search tree, the same B-move is a bad move at any lower level in the search tree.

B-moves have the virtue relative to T-moves that they do not increase the number of cards in the T-cells. B-moves either decrease or leave unchanged the number of cards in the T-cells, depending on whether the card moved comes from a T-cell or another B-column, respectively. T-moves always increase the number of cards in the T-cells, and dead-end positions in the Baker game are reached only when all T-cells are full. The Baker game program therefore makes B-moves prior to T-moves whenever both types of moves are available below a particular node. If more than one B-move is available, the multiple B-moves are chosen left-to-right. (Of course, neither B-moves nor T-moves are chosen if P-moves are available).

Efficacy of B-move Preference

B-move preference is almost as efficacious as P-move preference in reducing search time. For instance, consider a state S which has as successors $B_i(S)$ and $T_1(S)$ through $T_8(S)$. If $f(S)=0$, then $f(Q(S))=0$ for all successors $Q(S)$ of S. In this case it matters not the order in which the various successors of S are searched. It can only be hoped that the inevitable dead-ends below S are discovered as

quickly as possible. If on the other hand $f(S)=1$, then $f(Q(S))=1$ for at least one successor $Q(S)$ of S . In particular if $f(S)=1$, then the probability that $f(Bi(S))=1$ is very high. This is an empirical estimate rather than a provable fact, but the reduction in search time is significant in any case. Suppose for instance that if $f(S)=1$, then $p(f(Bi(S))=1)=0.5$. Even this modest result reduces the number of successors of S which must be searched approximately by half on the average whenever a B-move is available. If $p(f(Bi(S))=1)=0.9$, then the number of nodes which must be searched is reduced to less than two on the average whenever B-moves are available. The exact probability that $f(Bi(S))=1$ when $f(S)=1$ is not known, but it certainly seems rather high.

B-move preference is predicated on the hope that $f(Bi(S))=1$. But when a search proves that $f(Bi(S))=0$, the other aspect of B-move preference comes into play. The discussion of P-move preference included a proof that $Q_1Q_2\dots Q_nPi(S)=PiQ_1'Q_2'\dots Q_m'(S)$. A similar result holds for B-moves. Provided certain constraints are placed on all moves Q_j and Q_j' , it may be shown that $Q_1Q_2\dots Q_nBi(S)=BiQ_1'Q_2'\dots Q_m'(S)$ ($0\leq m\leq n$). When this equality holds, all states of the form $Q_1Bi(S)$, $Q_1Q_2Bi(S)$, and so forth through $Q_1Q_2\dots Q_nBi(S)$ may be eliminated from the search tree. That is, state $Q_1Q_2\dots Q_nBi(S)$ will be reached only if

$f(BiQ1'Q2'...Qm'(S))=0$, but $Q1Q2...QnBi(S)=BiQ1'Q2'...Qm'(S)$
so $f(Q1Q2...QnBi(S))=0$.

The conditions on the moves Q_j and Q_j' are roughly that they be independent of move Bi . For instance, consider a state $\underline{S}$ with successors $B1(S)$ which moves the 9-D from the second B-column to cover the 10-D on the first B-column and $T8(S)$ which moves the K-S from the eighth B-column to a T-cell. In this case, it is clear that $B1T8(S)=T8B1(S)$ and that $T8B1(S)$ can be eliminated from the search tree provided that $B1T8(S)$ is searched first. More generally, $BiQ(S)=QB_i(S)$ if Bi and $\underline{Q}$ are independent of each other.

The strictest definition of independence for a B-move Bi and any other move $\underline{Q}$ is that $\underline{Q}$ must neither add nor remove cards from the i -th B-column, and $\underline{Q}$ must neither move nor cover the card moved by Bi . If this definition of independence is met, it is clear by induction that $Q1Q2...QnBi(S)=BiQ1Q2...Qn(S)$ where Q_j is independent of Bi for all j , $1 \leq j \leq n$.

The conditions on Q_j and Q_j' may be relaxed further. By analogy to the proof of P-move preference, observe that $QB_i(S)=Bi(S)$ if move $\underline{Q}$ moves the same card as Bi . Thus, $\underline{Q}$ is a useless move and may be omitted. Therefore, removing from the requirements on Q_j and Q_j' the requirement that no move Q_j moves the same card as Bi leads by induction to the result that $Q1Q2...QnBi(S)=BiQ1'Q2'...Qm'(S)$ where each move

Q_j is independent of B_i or moves the same card as B_i for all j , and $Q_1'Q_2'\dots Q_m'$ is the sequence $Q_1Q_2\dots Q_n$ with all the moves omitted which move the same card as B_i ($m \leq n$). As before, $Q_1'Q_2'\dots Q_m'$ may be null.

Finally, the requirement that the card moved by B_i not be covered may be removed from the requirements on Q_j and Q_j' . Consider the sequence $Q_1Q_2\dots Q_nB_i(S)$ where all the moves Q_j are independent of B_i , move the same card as B_i , or cover the card moved by B_i . That is, the only requirement retained is the requirement that no move Q_j add to or remove cards from the i -th B-column. Suppose some move Q_k covers the card moved by B_i . This makes B_i impossible. Therefore, some move Q_m ($m > k$) must uncover the card moved by B_i . The process could repeat itself. There might be some move $Q_{k'}$ ($k' > m > k$) which covers the card moved by B_i again, making B_i impossible. Therefore, some move $Q_{m'}$ ($m' > k' > m > k$) must uncover the card moved by B_i , and so forth. In the end, no matter how many moves in the sequence $Q_1Q_2\dots Q_nB_i$ cover the card moved by B_i , some latter move in the sequence must undo the covering moves in order for B_i to be possible. It is therefore clear that $Q_1Q_2\dots Q_nB_i(S) = B_iQ_1'Q_2'\dots Q_m'(S)$ where the sequence $Q_1'Q_2'\dots Q_m'$ is the sequence $Q_1Q_2\dots Q_n$ modified as follows. All moves which move the same card moved by B_i are omitted as before. Then, all moves which cover the card moved by B_i are simply replaced by B_i . Finally, the moves

which uncover the card moved by B_i are replaced by T_i or E_i as appropriate.

This result may be clarified by an example. Suppose the following is a portion of a Baker state.

X-X	X-X	X-X
K-S	Q-S	J-S

Now, consider the sequence $Q_1Q_2B_1(S)=B_2T_2B_1(S)$. That is, the J-S is moved on top of the Q-S, the J-S is then moved to a T-cell, and finally the Q-S is moved on top of the K-S. Now, let $Q_1'=B_1$ and $Q_2'=T_1$. The new sequence is then $B_1Q_1'Q_2'=B_1B_1T_1$. That is, the Q-S is moved on top of the K-S, the J-S is moved on top of the Q-S, and finally the J-S is moved to a T-cell. The new sequence results in the same state as the first sequence, so $Q_1Q_2B_1(S)=B_1Q_1'Q_2'(S)$ and the sequence $Q_1Q_2B_1(S)$ may be omitted from the game tree. (In this example, the extra move of the J-S is clearly a useless move, but that is yet another strategy, to be considered later).

B-move/T-move Combinations

Just as a sequence T_iP_j may be considered as an inseparable entity, a sequence T_iB_j or T_iE_c may sometimes be considered to be an inseparable entity for the purposes of T-move reduction. Because of B-move preference, for any sequence T_iQ where Q is not a T-move it must be the case that move T_i made move Q possible. That is, if move Q were

available at the time T_i was made, Q would have been made rather than T_i ; therefore, Q was not available when T_i was made and T_i made Q available. In other words, move T_i uncovered a card which made a move available which was not available before. It may sometimes be the case that $T_i(T_jQ)(S) = (T_jQ)T_i(S)$, ($i < j$). When this equality holds, T_i for $i < j$ may be eliminated as a successor to $T_jQ(S)$. The conditions for the equality to hold are that move Q must not change the contents of the i -th B-column, and that Q must not remove a card from a T-cell. The implementation of this aspect of T-move reduction in the Baker game program involves the maintenance of the index which is used to scan the eight B-columns. The index is always set to 1 when a card is removed from a T-cell. Otherwise, it is set to the minimum of the indexes for the most recent T-move and the B-columns involved in move Q .

Empty B-columns

An E-move is a special case of a B-move which removes a card from the exposed end of a B-column and places it into an empty B-column. An I-move is a special case of a B-move which removes a card from a T-cell and places it into an empty B-column. As a result, E-moves and I-moves cannot be made from an initial Baker state, nor can they be made early in a Baker game. Prior to the exposure of an empty

B-column, only P-moves, ordinary B-moves, and T-moves may be made. At the time an empty B-column is first exposed, the game must already be fairly mature and the probability that the game may be won is fairly high.

The strategies already developed for P-moves, B-moves, and T-moves, plus a few more which have not yet been developed, tend to make an exhaustive Baker game search procedure seem plausible. The occurrence of an empty B-column, on the other hand, causes the potential size of the search tree to increase radically, casting doubt on the plausibility of an exhaustive search solution for the game. When a B-column first becomes empty, there are 7 E-moves available (moving a card from any one of the other B-columns to the empty B-column), there may be up to 4 I-moves available, there are probably 7 T-moves available, and there may be one or two B-moves available. There may also be a P-move available. If so, the P-move preference strategy as always causes the P-move to be made immediately and defers the problem of dealing with an empty B-column to one level lower in the search tree. In any case, empty B-columns create a situation in which a node has as many as 20 or so successors.

Success in dealing with empty B-columns hinges on two factors. One is the empirically determined high probability that a Baker state with an empty B-column is a winning

state. The other is the similarities of E-moves and I-moves to other types of moves.

A state with an empty B-column is intuitively a desirable state, one which is likely to be winnable. At a minimum, an empty B-column acts like an extra T-cell, serving to store one more card than usual. At a maximum, an empty B-column may be used as the base for a string of cards, possibly beginning with a king, queen, and jack, or other high (and thereby "difficult") card.

When a single card is moved into an empty B-column from another B-column, the E-move is almost identical to a T-move, and strategies must be developed to exploit the similarities between E-moves and T-moves. When multiple cards are placed into an empty B-column, an E-move is almost identical to a B-move, and strategies must be developed to exploit the similarities between E-moves and B-moves.

When a single card is moved into an empty B-column from a T-cell, the I-move is almost like no move at all, and strategies must be developed to exploit the similarity of the I-move to an identity operator. When multiple cards are placed into an empty B-column, an I-move is almost identical to a B-move, and strategies must be developed to exploit the similarities between I-moves and B-moves.

The difficulty in developing such strategies is that it is not in general known at the time the first card is placed

into an empty B-column whether or not it will be followed by other cards.

The discussion of empty B-columns as presented thus far (pessimistic though it may be) actually understates the difficulties which empty B-columns introduce into a search tree for the Baker game. After empty B-columns first appear, they tend to persist to lower levels of the tree, possibly for many levels. The difficulty lies in the ease with which cards may be exchanged between T-cells and empty B-columns. T-moves, E-moves, and I-moves may follow each other in long sequences. Empty B-columns do not necessarily appear at every node in the sequence, but they do appear intermittently throughout the sequence. Empty B-columns thus produce not just an isolated abnormal node with as many as 20 successors, but may produce many nodes at various places in a sequence of moves which may have as many as 20 successors. Therefore, when a long sequence of nodes is searched where a number of the nodes contain empty B-columns, it is easy for the program to "spin its wheels", making neither progress toward winning the game nor progress toward the identification of losing states.

Reversible Moves

The difficulties previously discussed with regard to empty B-columns are a subset of a slightly larger problem

which might be called the reversible move problem. Many Baker game moves result in a state in which the card just moved may be legally returned to its original location on the very next move. That is, given a state $\underline{S}$ with successor $Q1(S)$, there is often a move $Q2$ such that $Q1Q2(S)=S$. Reversible moves of course lead to an infinite search tree. For instance, if $Q1Q2(S)=S$, then $Q1Q2Q1Q2(S)=S$, $Q1Q2Q1Q2Q1Q2(S)=S$, etc. Even if the infinity is somehow eliminated, reversible moves increase the number of moves available at each node rather severely.

P-moves are never reversible, but for all other moves the reversibility of the move is determined by the location from which the card is moved. For the purposes of determining reversibility, a movable card may be in one of four positions. A movable card is one which is in a T-cell, one which is the only card in a B-column, an exposed card which covers the next higher card of the same suit in a B-column, and an exposed card which covers a card other than the next higher card of the same suit in a B-column. In the first three cases, a move of the indicated card is a reversible move and in the fourth case a move of the indicated card is not reversible.

The obvious strategy of not reversing a reversible move on the very next move is not adequate to cope with reversible moves. For instance, suppose in some state $\underline{S}$

that the J-C is on top of the Q-C in the eighth B-column, and that the fourth B-column and at least one T-cell are empty. Then, the J-C may be moved freely between the fourth B-column, the eighth B-column, and a T-cell essentially forever. For example, T8B8(S)=S, E8T4B8(S)=S, and so forth.

The problem of reversible moves is compounded by the fact that there may be more than one card which floats in the manner just described for the J-C. A long sequence of moves with multiple floating cards may be made without ever making any progress towards winning the game or identifying losing states. Of course, the moves of the floating cards may be intermixed with moves of other cards, making identification of essentially equivalent but unequal states difficult. Finally, moves of floating cards may not be eliminated a priori because moves of these cards may be necessary to a winning solution of the game.

It is first necessary to eliminate recursions which may result from reversible moves. Consider a state S which is followed by states $Q_1(S)$, $Q_1Q_2(S)$, $Q_1Q_2Q_3(S)$, etc. where each move of the sequence $Q_1, Q_2, \dots, Q_n$ is reversible. Rename the first state S as S_0 , the state $Q_1(S)$ as S_1 , the state $Q_1Q_2(S)$ as S_2 etc. Each time a reversible move Q_j is made where each Q_i ($i < j$) is also reversible, compare state S_j with each state S_i ($0 \leq i < j$). If $S_i = S_j$ for some i , then a

recursion has occurred. When a recursion occurs, eliminate move Q_j as a successor of state $S(j-1)$.

This strategy assures that the Baker search tree is finite. It also preserves the exhaustive nature of the search because the states which are eliminated (and all of their descendants) are known still to exist at a higher level in the tree.

The elimination of recursion does not contribute very much in some respects to the production of a practical exhaustive search program for the Baker game. Even with recursions eliminated, reversible moves increase the number of successors below each node so much that an exhaustive search is impossible without additional strategies to reduce the search time.

Successive Move Suppression

The first strategy to be developed for reversible moves may be called successive move suppression. It is the simplest: do not move the same card two times in succession. There are two cases to consider in validating this strategy. The second move of the same card may either return the card to its original location, or it may move the card somewhere else. In the former case, if Q_1 and Q_2 move the same card, then $Q_1Q_2(S)=S$ and Q_2 may clearly be eliminated as a successor of $Q_1(S)$. In the latter case, $Q_1Q_2(S)=Q_2(S)$ and

Q2 may be eliminated as a successor of $Q1(S)$ because the same state exists one level higher in the tree as state $Q2(S)$.

All remaining strategies to deal with reversible moves are generalizations on the idea of moving a given card only a single time in succession. That is, consider a sequence of moves $R1Q1Q2...QnR2(S)$ where $R1$ and $R2$ move the same card. Depending on the situation, there may be a sequence of moves $Q1'Q2'...Qn'$ related to the sequence $Q1Q2...Qn$ such that $R1Q1Q2...QnR2(S) = Q1'Q2'...Qn'(S)$ or $R1Q1Q2...QnR2(S) = R2Q1'Q2'...Qn'(S)$. In other words, despite intervening moves, moving the same card two different times may either return it to its original location or move it on the second move the same place it could have been moved on the first move.

B-move Suppression

The first generalization on successive move suppression may be called B-move suppression. In this case, $R1$ removes a card from the next higher card of the same suit in a B-column i and $R2$ returns the card to the i -th B-column. Clearly, $R1R2(S) = S$. The conditions under which $R1Q1Q2...QnR2(S) = Q1'Q2'...Qn'(S)$ are as follows:

- 1) All states $R1Q1Q2...Qk(S)$ for $1 < k < n$ must contain the same number of cards in the i -th B-column.

- 2) All moves Q_j are replaced by $Q_j' = Q_j$, where all the moves Q_j and Q_j' are considered to be indexed by card rather than by location.

The implementation of B-move suppression in the Baker game program consists of refusing to return a card to a B-column from which it was removed unless the number of cards in the B-column is changed in the interim.

E-move Suppression

E-move suppression is similar to B-move suppression except that move R1 removes a card from the i -th B-column leaving it empty, and R2 returns the card to an empty B-column. The conditions required to satisfy E-move suppression are the same as for B-move suppression with one slight relaxation. Condition 1) for B-move suppression would imply for E-move suppression that the i -th B-column must remain empty for all states $R_1 Q_1 Q_2 \dots Q_k(S)$. This condition may be relaxed to the extent that at least one (not necessarily the i -th) B-column must remain empty for all states $R_1 Q_1 Q_2 \dots Q_k(S)$, and that R2 may return the card to any (not just the i -th) empty B-column. The implementation of E-move suppression in the Baker game program consists of refusing to return a card to an empty B-column if all interim states include an empty B-column.

T-move Suppression

T-move suppression is similar to B-move suppression and

E-move suppression except that move R_1 removes a card from a T-cell and move R_2 returns it to a T-cell. The conditions under which $R_1Q_1Q_2\dots Q_nR_2(S)=Q_1'Q_2'\dots Q_n'(S)$ are as follows for T-move suppression:

- 1) All states $R_1Q_1Q_2\dots Q_k(S)$ for $1\leq k\leq n$ must contain at least one empty T-cell.
- 2) No move Q_j may cover the card moved by R_1 .
- 3) All moves Q_j are simply replaced identically by $Q_j'=Q_j$.

B-move suppression, E-move suppression, and T-move suppression are all generalizations of $R_1R_2(S)=S$. The generalizations of $R_1R_2(S)=R_2(S)$ follow. They are called B*-move suppression, E*-move suppression, and T*-move suppression.

B*-move Suppression

B*-move suppression occurs when R_1 moves a card from a B-cell other than from the next higher card of the same suit, from an E-cell, or from a T-cell, and when R_2 is a B-move which moves the same card to the i -th B-cell. The conditions under which $R_1Q_1Q_2\dots Q_nR_2(S)=R_2Q_1'Q_2'\dots Q_n'(S)$ for B*-move suppression are as follows:

- 1) $R_2(S)=B_i(S)$ is a legal move.
- 2) The number of cards in the i -th B-column is the same for all states $R_1Q_1Q_2\dots Q_k(S)$, $1\leq k\leq n$.

- 3) All moves which cover the card moved by $R1$ are replaced by $Qj'=Bi$; otherwise, $Qj'=Qj$.

B*-move suppression is implemented implicitly in the Baker game program. B-moves are made prior to E-moves, I-moves, and T-moves. Making B-moves first plus normal B-move suppression assures that a card will never be played to a B-cell later than would be possible.

E*-move Suppression

E*-move suppression occurs when $R1$ moves a card from a B-column or from a T-cell and when $R2$ moves the same card to an empty B-column. The conditions under which $R1Q1Q2...QnR2(S)=R2Q1'Q2'...Qn'(S)$ for E*-move suppression are as follows:

- 1) $R2(S)=Ei(S)$ is a legal move.
- 2) There is at least one empty B-column for all states $R1Q1Q2...Qk(S)$, $1 \leq k \leq n$.
- 3) All moves which cover the card moved by $R1$ are replaced by $Qj'=Bi$; otherwise, $Qj'=Qj$.

E*-move suppression is implemented explicitly in the Baker game program. When an E-move is to be made, previous moves of the card making the E-move are examined. If there is a previous move of the same card satisfying E*-move suppression, the move is not made.

T*-move Suppression

T*-move suppression occurs when $R1$ moves a card from a

B-column to another B-column, and when R_2 is a T-move. The conditions under which $R_1Q_1Q_2\dots Q_nR_2(S) = R_2Q_1'Q_2'\dots Q_n'(S)$ for T*-suppression are as follows:

- 1) $R_2(S) = T_k(S)$ is a legal move, where R_1 moves a card from column B_k to B_i .
- 2) There is at least one empty T-cell for all states $R_1Q_1Q_2\dots Q_k(S)$, $1 \leq k \leq n$.
- 3) No move Q_j may cover the card moved by R_1 ; each Q_j is replaced identically by $Q_j' = Q_j$.

T*-move suppression is implemented explicitly in the Baker game program. When a T-move is to be made, previous moves of the card making the T-move are examined. If there is a previous move of the same card satisfying T*-move suppression, the move is not made.

The various problems with reversible moves occur toward the end of the game. A similar problem also occurs toward the end of the game. The nature of the game tends to create runs of cards of the same suit in a given B-column. For instance, late in a game a B-column might contain a run consisting of the 9-D, 8-D, 7-D, and 6-D.

The first problem with runs of cards in a B-column is that they can lead to many extra moves where the moves simply dead-end. If cards from a run are moved to the T-cells, they will simply fill the T-cells and create a dead-end unless there is at least one empty B-column, unless

the highest ranking card in the run can be uncovered and moved to a B-column, or unless the entire run can be moved to the T-cells uncovering another card or emptying a B-column. In case a card not in the run is uncovered, there is still a dead-end unless the uncovered card can be moved or serve as the target of a B-move. This particular situation, despite the apparent complexity of its description, is quite simple to recognize and is quite common. The strategy which anticipates dead-ends when moving a run is called MOVRUN Suppression. The MOVRUN suppression strategy is implemented in the Baker game program as logic in the move selection routines. The logic will remove cards from a run only if a dead-end will not occur at the end of the run.

The second problem with runs of cards in a B-column is that they tend to be able to reach a given state with many different sequences of moves. Consider a sequence of moves $T_k Q_1 Q_2 \dots Q_n R(S)$ where T_k removes a card from a run, none of the Q_i moves the card moved by T_k or uncovered by T_k , and R moves the card uncovered by T_k . Then, it is obvious that $T_k Q_1 Q_2 \dots Q_n R(S) = Q_1 T_k Q_2 \dots Q_n R(S)$ and so forth through $T_k Q_1 Q_2 \dots Q_n R(S) = Q_1 Q_2 \dots Q_n T_k R(S)$. In other words, T_k is independent of all moves Q_i and may be made in any order relative to them. The condition that no move Q_i move the same card moved by T_k is enforced by B*-suppression and

E*-suppression. Therefore, the interpretation of the equations above is that when a T-move removes a card from a run, that T-move can be made at any time provided it is made by the time it is desired to move the uncovered card. The Baker game program makes the choice of making the moves only in the order $Q_1Q_2\dots Q_nTkR(S)$. The implementation of this strategy consists of always moving the uncovered card on the very next move when a T-move removes a card from a run.

A similar strategy is used when E-moves remove a card from a run. Consider a sequence of moves $E_kQ_1Q_2\dots Q_nR(S)$ where E_k removes a card from a run, none of the Q_i moves the card moved by E_k or uncovered by E_k , none of the Q_i covers the card moved by E_k , and R moves the card uncovered by E_k or covers the card moved by E_k . Then, E_k may be made before or after any Q_i and the Baker game program makes the moves only in the order $Q_1Q_2\dots Q_nE_kR(S)$. This strategy consists of always moving the uncovered card or covering the moved card on the very next move when an E-move removes a card from a run.

The third problem with runs of cards in a B-column occurs when a run covers a card not in the run, and the run is moved to uncover the card not in the run. For instance, the 9-D, 8-D, 7-D, and 6-D might cover the 2-C. The program tends to function crisply, with few wasted moves, when a run is moved to the next higher card of the same suit to make a

new, longer, run. In this case, it would tend to move the run crisply to cover the T-D, using the T-cells and possibly some empty B-columns as temporary storage for the cards in the run while it is being moved. However, when a run is moved to T-cells and empty B-columns, the cards from the run can be distributed in many different ways at the point where the last card in the run is removed.

It can be shown that moving as many cards as possible of a run into one B-column is the optimal way of moving the run. The informal rationale for this result is that moving the maximum number of cards to one B-column stores the maximum number of cards in the minimum amount of space, leaving as many B-columns and T-cells free as possible. Moving any fewer cards creates a state occupying more space, ultimately leaving fewer empty B-columns and T-cells, thereby reducing the number of paths which can be taken through the search space. The proof of this result is as follows:

Theorem: Suppose state S has a run of length n , and suppose a series of moves R_k moves k of the cards in the run to an empty B-column $\bar{\quad}$ ($1 \leq k < n$). Then, $f(R(k+1)(S)) \geq f(R_k(S))$ for $1 \leq k < n$.

Proof: If $f(R_k(S)) = 0$, then the inequality clearly holds. Therefore suppose $f(R_k(S)) = 1$. In other words, suppose there is some sequence of moves $Q_1 Q_2 \dots Q_n$ such that $R_k Q_1 Q_2 \dots Q_n(S)$ is the goal state. It is required to find a sequence of moves $Q_1' Q_2' \dots Q_m'$ such that $R(k+1) Q_1' Q_2' \dots Q_m'(S)$ is the goal state given that $R_k Q_1 Q_2 \dots Q_n(S)$ is the goal state. There are two cases to consider. 1) If no move Q_i covers card $k+1$ of the

run, then $R(k+1)Q_1'Q_2'\dots Q_m'(S)$ is clearly the goal state where $Q_1'Q_2'\dots Q_m'$ is the sequence $Q_1Q_2\dots Q_n$ with all moves omitted which move card $k+1$ of the run except the P-move for card $k+1$. 2) If at least one move Q_j covers card $k+1$ of the run, the situation is more complex. Consider the sequence $R_kQ_1Q_2\dots Q_{(j-1)}Q_jQ_{(j+1)}\dots Q_n(S)$, where Q_j covers card $k+1$ of the run. In order to make move Q_j , the moves Q_i for $1 \leq i < j$ must basically have undone the R_k sequence, removing the cards from the B-column where the R_k sequence of moves placed them. Because the moves Q_i which undo R_k are themselves reversible, the entire run moved by R_k could be placed on the same B-column as card $k+1$ with the Q_i moves for $i > j$. Because these moves are all reversible, the sequence $R_kQ_1Q_2\dots Q_{(j-1)}Q_jQ_{(j+1)}\dots Q_n(S)$ can be replaced by the equivalent sequence $R_kQ_1'Q_2'\dots R(k+1)\dots Q_m'(S)$, where the Q_i' notation is required because the moves must be renumbered. The moves of $R(k+1)$ after Q_j may or may not exist as moves Q_i ($i > j$). If they exist, they may be gathered next to Q_j . If they do not exist, they and subsequently their inverses may be inserted because they are reversible. The moves of $R(k+1)$ which undo R_k are independent of the moves Q_i ($i < j$) and be made in any order relative to them. In particular, they may be gathered next to Q_j . The initial R_k sequence thus is redundant and may be omitted, leaving the equivalent sequence $Q_1'Q_2'\dots R(k+1)\dots Q_m'(S)$. Finally, the $R(k+1)$ sequence is independent of all the moves Q_i' and may be made in any order relative to them. Thus, the $R(k+1)$ sequence may be moved to the beginning of the overall sequence leaving $R(k+1)Q_1'Q_2'\dots Q_m'(S)$ as a sequence equivalent to $R_kQ_1Q_2\dots Q_n(S)$. QED

Because of this result, the program always distributes the cards in just one way. If there are no empty B-columns, the cards in the run are all placed in T-cells. If there are empty B-columns, as many of the cards as possible in the run are placed in the first empty B-column, as many of the residual cards as possible are placed in the second empty B-column, and so forth through placing the final residual cards in T-cells. If there are B empty B-columns ($B \geq 1$) and

T empty T-cells, then $(T+1) \cdot [2^{B-1}]$ cards may be moved to the first empty B-column. The total number of cards that may be moved to all the empty B-columns is $(T+1) \cdot [2^{B(B-1)/2}]$, and the total number of cards that may be moved to all the empty B-columns and T-cells is $(T+1) \cdot [2^{B(B-1)/2} + 1] - 1$.

The distribution of cards from a run is performed by a recursive algorithm much like the algorithm to solve the classical Tower of Hanoi puzzle. The algorithm moves as many of the cards of the run as possible and compresses them into as few new runs as possible (usually just one) in the empty B-columns.

CHAPTER III

THE BAKER GAME PROGRAM

Programming Language

The Baker game program is written in FORTRAN IV. The program was compiled with the IBM FORTRAN Q compiler at optimization level 3. The IBM FORTRAN Q compiler is the enhanced version of the IBM FORTRAN X compiler, which in turn is the extended version of IBM FORTRAN H compiler. The program was run on IBM 3031 computers running the SVS operating system at the University of Tennessee Computing Center.

FORTRAN was chosen because of its portability. A major weakness of FORTRAN is that it does not support recursion, and the search procedure used by the Baker game program is inherently stack oriented. Nevertheless, it is not difficult to simulate recursion in FORTRAN using a user-controlled stack. Also, some of the program's strategies require access to the stack contents, and such access is easy in a user controlled stack.

Program Management

The Baker game program is modularized into a number of FORTRAN subroutines. Sharing of data across subroutines is achieved via FORTRAN COMMON blocks. Debugging was achieved

via insertion of FORTRAN WRITE statements before every decision point in the program. The WRITE statements identify the decision point and print the value of every variable that was assigned a value since the previous decision point. In order to assist in management of the COMMON blocks and in management of the WRITE statements used for debugging, a preprocessing program was written. The preprocessing program inserts the same COMMON blocks into each subroutine before compilation. Thus, changes in the data areas are made only one time and are then automatically inserted into each subroutine. Also, the preprocessing routine will optionally remove the WRITE statements used for debugging from each subroutine before compilation. Removal of the debugging statements was important for reasons of performance. All the debugging statements were conditional, and could be enabled or disabled with appropriate control cards. However, it was found that the program executed about twice as fast with the debugging statements removed as with the debugging statements remaining in the program, even when the debugging statements were not executed.

Data Structures

There are three structures in the game that are supported by data areas in the program. These structures are the card deck, the playing board, and the game tree.

The card deck is represented by a series of vectors of dimension 52 to contain the attributes of the 52 cards. For instance, `MOVBLE(N)` is a LOGICAL (Boolean) variable that specifies whether or not the n-th card is currently movable. The playing board is represented by a series of vectors and one matrix that define the P-cells, B-columns, and T-cells. The dimensionality of these variables is appropriate to the kind of cell involved. For instance, `BCELLS(I,J)` contains the j-th card in the i-th B-column, `IBCELL(I)` contains the number of cards currently stored in the i-th B-column, and `TCELL(I)` contains the card in the i-th T-cell. Finally, the game tree is represented by a series of vectors of dimension 300 which represent the search tree. The search procedure is a generative depth-first procedure, so that only the path from the initial state to the current path is maintained in memory. The vectors of dimension 300 permit search depths up to level 300, but winnable games seldom extend beyond level 150 or so, and unwinnable games seldom extend beyond level 50 or so.

In addition to the data areas required for the representation of the game, there are a number of miscellaneous data areas. They are generally either scalars, or else are vectors specific to some particular subroutine or some specialized function. For instance, the subroutine that detects recursions of reversible moves has a

vector to store each state in a sequence of reversible moves.

Data Elements

The following is a list of the data elements used in the program, including a description of their functions.

R(N) is a pseudo-random number associated with the n-th card. It is used with sorting and shuffling the cards.

P1(N) contains the lookup index for the DUPCHK routine when the B-columns contain n cards.

P2 is a 15,000 element vector used to save the lookup lists and copies of the states of the game in the DUPCHK routine.

RD is the FORTRAN input unit number (5 for IBM systems).

T1 through T8 contain 16**0 through 16**7 as packing and shifting constants for the DUPCHK routine.

WR is the FORTRAN output unit number (6 for IBM systems).

BMAP is an 8 element vector used to re-order the 8 B-columns for the DUPCHK routine and the RVDPCK routine.

CARD is a 52 element vector used to generate the rank function for each card. CARD(N) contains the rank (1 through 13) for the n-th card.

HEUR is a flag indicating that heuristics were used in the solution of the game.

MOVE(M) contains the identity of the card moved on the m-th move.

MXP2 contains the size of the P2 vector.

NULL is an error flag indicating there are no control cards for the program.

OLDI is used to save and restore the index to the current B-column when B-moves are made.

PASS is a flag indicating which of the 4 passes the game is playing when playing the game multiple times with different strategies.

SUIT is a 52 element vector used to generate the rank function for each card. SUIT(N) contains the suit (1 through 4) for the n-th card.

ABORT is a flag used to terminate the program for timer overflows and various error conditions.

AVAIL is an 8 element vector which supports B-move suppression. AVAIL(I) is .FALSE. only if B-move suppression is in effect for the i-th B-column.

BSAVE is a 20,000 element vector used for saving the contents of the B-columns for the use of the RVDPCK routine.

CARDY is a 52 element vector used to contain the deck for shuffling and dealing. CARDY(N) contains the n-th card in the deck in shuffled order.

FREEB(M) contains the number of empty B-columns on the m-th move.

FREET(M) contains the number of empty T-cells on the m-th move.

GENOP is a flag indicating whether the initial pseudo-random number for the deck shuffling was supplied on a control card or should be generated internally.

LASTN is the identity of the most recently moved card.

MVCNT is the current level in the search tree.

MVIND(M) is an index indicating the location from which a card is moved on the m-th move. It is the value of WHIND(N) before card n is moved on the m-th move.

MVNUM(M) is an index indicating the location from which a card is moved on the m-th move. It is the value of WHNUM(M) before card n is moved on the m-th move.

NDEAL is the number of times the program is to play the game.

NZERO is the number of empty B-columns.

OLDEL is used to save and restore the index to the B-column most recently used as a source for E-moves.

PCELL(I) is the i-th P-cell.

P2CNT is the current index into the P2 vector in the DUPCHK routine.

RINIT is the initial seed for the pseudo-random number generator.

SHORT is a flag used by the E-move and T-move suppression routines to indicate that the proposed move is not to be made.

TOIND(M) is an index indicating the location to which a card is moved on the m-th move. It is the value of WHIND(N) after card n is moved on the m-th move.

TONUM(M) is an index indicating the location to which a card is moved on the m-th move. It is the value of WHNUM(N) after card n is moved on the m-th move.

WHERE(N) is an index indicating the location of the n-th card (1 for P-cells, 2 for B-columns, and 3 for T-cells).

WHIND(N) is an index indicating the location within the P-cell (1 through 13), B-column (1 through 19), or T-cell (always 1) which contains card n.

WHNUM(N) is an index indicating the particular P-cell (1 through 4), B-column (1 through 8), or T-cell (1 through 4) which contains card n.

BCELLS(I,J) contains the j-th card in the i-th B-column.

BFROZE is an 8 element vector used to implement the B-frozen heuristic (see Chapter IV).

BSAVPT is a 300 element vector used to index the BSAVE vector in the RVDPCCK routine. BSAVPT(M) contains the first location in BSAVE used on the m-th move.

BSVCNT is the index to the current location being accessed in the BSAVE vector.

FORCED(M) is a flag indicating that the m-th move was made by the MOVRUN subroutine.

FORCEN is a flag used when unstacking a run from a B-column. It is set equal to the uncovered card.

FORCE2 is a flag used when unstacking a run from a B-column. It is set equal to the next lower card than the one moved when a card from a run is unstacked to an empty B-column.

IBCELL(I) contains the number of cards in the i-th B-column.

ITCELL contains the number of cards in the T-cells.

LEMPY is an index to the left-most empty B-column.

MOVBLE(N) is a flag indicating whether or not card n is movable.

MOVUSD is a flag indicating whether or not the MOVRUN subroutine was used in the solution of the game.

MVTIME(M) contains the cumulative machine time spent up until the m-th move.

MVWHERE(M) is an index indicating the location from which a card is moved on the m-th move. It is the value of WHERE(N) before card n is moved on the m-th move.

NBCELL is the number of B-columns in use (6 for a two suit game, 7 for a three suit game, and 8 for a four suit game).

NCARDS is the number of cards in the deck (26 for a two suit game, 39 for a three suit game, and 52 for a four suit game).

NEWPGE is a flag used to request a new page when subroutine SEE is invoked.

ONLY1 is the number of B-columns containing only one card.

NSHUFF is the number of times the deck is shuffled before each game.

NSUITS is the number of suits to be used for each game.

NTCELL is the number of T-cells used for each game (set equal to NSUITS).

OLDELS(M) is used to save the value of OLDEL on the m-th move.

OVRFLO is a flag indicating that the RVDPC routine ran out of memory.

PACKED(M) contains a representation (8 values combined into one location) of the number of cards in each of the 8 B-columns on the m-th move.

PCOUNT(M) is the number of cards in the P-cells on the m-th move.

PLEVEL is a flag indicating the level of detail of the printout from the program.

PREVMV(N) is the index to the last move made by the n-th card.

READOP is a flag indicating whether or not the program is to read the card deck from control cards.

RESULT is a flag used to indicate whether the game was won, lost, or undecided.

REVERS(M) is a flag indicating whether or not the m-th move was reversible. A value of 1 is used to indicate non-reversible moves. A value of k greater than 1 indicates that k-1 consecutive reversible moves were made connecting k states.

SAVAIL(M) is used to save the value of AVAIL(N) when card n is moved on the m-th move.

SBFROZ(M) is used to save the value of BFROZE on the m-th move.

SEEMOV is an index used to indicate to subroutine SEE which state to display.

SFORCN(M) is used to save the value of FORCEN on the m-th move.

SFORC2(M) is used to save the value of FORCE2 on the m-th move.

SRTCNT is an 8 element vector used to contain and sort the number of cards in each B-column in the BSORT routine.

SRTC RD is an 8 element vector used to contain and sort the first card in each B-column in the BSORT routine.

STFROZ(M,*) is used to save the values of TFROZE(1) through TFROZE(4) on the m-th move.

TIMEOP is a flag indicating whether or not program timing is performed.

TIMSAV is a 21 element vector used to save the last time that each of the 21 clocks was started.

TIMSUM is a 21 element vector used to accumulate time for 21 separate clocks.

TMOVES(M) is the total number of nodes examined as of the m-th move in the current path.

TMVCNT is the current value for the total number of nodes examined.

TOWHER(M) is an index indicating the location to which a card is moved on the m-th move. It is the value of WHERE(N) after card n is moved on the m-th move.

ZMVCNT is the maximum level examined in the search tree.

Program Routines

The following is a list of the routines used in the program, and a synopsis of their function.

BAKER is the main routine, consisting primarily of subroutine calls. It is the dispatcher and control routine for the program.

BAKTRK removes the most recent move from the current path when a dead-end occurs. It is a mirror image of MVCARD.

BLOCK is a FORTRAN BLOCK DATA subprogram with DATA statements for program constants.

BSORT is a sort routine used to sort the B-columns for duplicate checking in subroutines DUPCHK and RVDPCCK.

CHECK determines if the next move will create an immediate dead-end. It is invoked as a part of the neighborhood reduction logic when all but one T-cell is full.

DEAL deals the cards from the CARDY vector into the BCELLS matrix.

DUPCHK was an early and only marginally successful attempt to solve the end game problem. It was largely superseded by the various B-move and T-move suppression strategies. It saves certain key states (states with large branching factors), and then attempts to determine if newly encountered states have occurred previously. It uses an indexed, chained list structure to make the search of the previous states as fast as possible. It was retained in the program because it does work correctly and does provide some marginal benefit, even though the reduction in tree size realized by utilizing this routine was not as great as desired.

FIRST is an assembler language routine which generates an initial seed for the random number generator using the low-order bits of the system clock.

INIT performs initialization functions on the card deck. It either generates the deck or reads it in from control cards, and then shuffles it.

MOVRUN moves a run of cards from a B-column to another B-column.

MVCARD makes one move and updates all the requisite variables. It is a mirror image of BAKTRK. MVCARD and BAKTRK are the main bookkeeping routines in the program.

PLAY is the main game playing routine in the program. It decides which moves to make, and calls the MVCARD and BAKTRK routines as required to make those moves.

QKRSRT is a sort routine used by the SHUFFL routine.

RANDU is a standard system pseudo-random number generator provided by IBM with the OS family of operating systems.

READER reads program options and sets flags based on the input control cards.

REVIEW is used to print out the winning solution to a game after the game is completed.

RVDPCCK prevents infinite loops by detecting recursions when reversible moves are made.

SEE prints the current state of the game.

SHORTN implements the strategies for E-move, T-move, E*-move, and T*-move suppression.

SHUFFL shuffles the deck by associating a pseudo-random number with each card in the deck and sorting the cards according to the pseudo-random numbers.

SSTATE is used to save key states of the game for use by the DUPCHK routine.

TIMER is used to maintain clocks to time various parts of the program.

ZERO is used to initialize program variables to zero, or to the appropriate null values.

Program Options

The program supports a number of options that are read from control cards. It will continue reading cards and playing games until the deck of control cards is exhausted. The available options permit selection of a two, three, or four suit game, permit specification of the number of times the game is to be played and the number of times the deck is shuffled before each game, and permit initial random numbers

and initial decks to be specified in a number of ways. The following is a list of the options.

GENOP (column 1). GENOP=1 indicates that a particular game is to be solved (see READOP below), and GENOP=0 indicates that a game or series of games is to be generated randomly and solved. The default is GENOP=0 when column 1 is blank.

PLEVEL (column 2). PLEVEL indicates the level of detail concerning the activities of the program which is to appear in the printed output. See the description of subroutine SEE later in this chapter.

READOP (column 3). READOP=1 indicates that a deck of 52 cards, each containing a number from 1 through 52 in the first two columns, follows the control card and is to be used as the initial deck. READOP=0 indicates that the initial deck is generated by the program. The default is READOP=0 when GENOP=0, but READOP=1 is forced when GENOP=1.

NSHUFF (columns 4-5). NSHUFF is the number of times the deck is shuffled before each game.

NDEAL (columns 6-10). NDEAL is the number of times the game is to be played. The default is NDEAL=1 when columns 6-10 are blank.

TIMEOP (column 11). TIMEOP=1 indicates that no program timing is to occur, and TIMEOP=0 indicates that program timing is to occur. The default is TIMEOP=0 when column 11 is blank.

NSUITS (column 12). NSUITS is the number of suits, and thereby the number of P-cells and T-cells, to use. The default is NSUITS=4 when column 12 is blank.

RINIT (columns 13-22). RINIT is the initial value of the seed for the random number generator. The default value is generated randomly from the low-order bits of the system clock when columns 13-22 are blank.

General Program Operation

The overall operation of the program is as follows. READER is invoked to read program options. After the options are read, the INIT, SHUFFL, QKRSRT, ZERO, and DEAL routines are invoked to initialize, shuffle, and deal the deck, and to initialize the program's data areas. Finally, the PLAY routine is invoked to play the game.

The overall operation just described may be displayed graphically as shown in Figure 3 below:

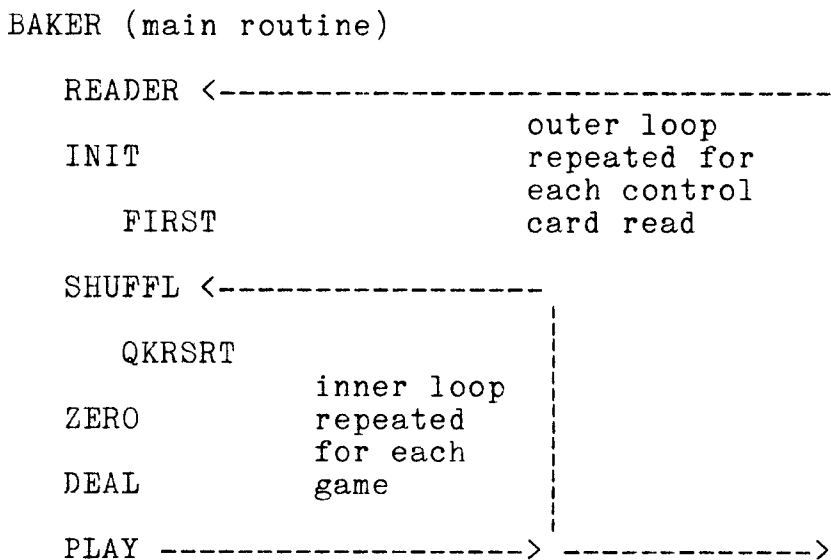


Figure 3

The PLAY Routine

The PLAY routine decides which moves to make and invokes the MVCARD and BAKTRK routines to make those moves. There are four major sections of the PLAY routine, one to make P-moves, a second to make B-moves (including E-moves and I-moves), a third to make T-moves, and a fourth to backtrack. Each of the first three sections determines a move that can be made and calls MVCARD. When the P-move section cannot make a move, it goes to the B-move section. When the B-move section cannot make a move, it goes to the T-move section. When the T-move section cannot make a move, it goes to the backtrack section to remove the previous move from the game tree. The internal structure of the PLAY routine may be displayed as shown in Figure 4.

PLAY

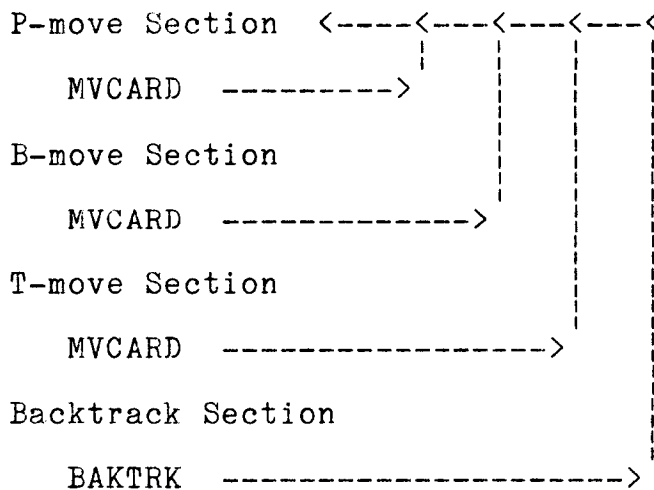


Figure 4.

Each of the first three sections has pre-move processing to decide which move to make, and post-move processing to determine if the move just made creates a duplicate state or other state for which it is known that no further searching is required. Pre-move processing is contained primarily within the PLAY routine. The processing within PLAY is supplemented by the CHECK and SHORTN routines. The CHECK routine implements the dead-end detection strategy. The SHORTN routine implements the various B-move and T-move suppression strategies. Post move processing is contained in the DUPCHK and RVDPCCK routines. These two routines test for duplicate game states in the general case (DUPCHK) and in the special case of reversible moves (RVDPCCK). The DUPCHK routine uses the BSORT routine to sort the B-columns, and the SSTATE routine to save the current state in a lookup table.

The P-move section is a common return point for each of the sections. As such, it tests for termination criteria. The termination criteria are: 1) game won, 2) game lost, and 3) time excession. In the case of games which are won or games with time excession, the REVIEW routine is called to print out the winning solution to won games or the current status of games for which the time was exceeded.

SEE Routine

The SEE routine displays the state of the game in a form that looks as much as possible like an actual card deck dealt for the Baker game. It is a service routine that is called from a large number of locations in the program. Depending on the requested option, SEE may be invoked after the cards are dealt by DEAL, during the game by MVCARD and BAKTRK, and after the game by REVIEW. The program option that controls the SEE routine is called PLEVEL. PLEVEL=1 simply prints the initial game state and whether the game was won or lost. PLEVEL=2 adds an abbreviated summary of the solution for winnable games. PLEVEL=3 adds a detailed summary of the solution for winnable games by using the REVIEW and SEE routines. PLEVEL=4 causes the SEE routine to print each state encountered in the MVCARD routine (when the program is searching forward). PLEVEL=5 causes the SEE routine to print each state encountered in both the MVCARD and BAKTRK routines (when the program is searching forward, and when it is backtracking). PLEVEL's greater than 5 are used for debugging.

Debugging and Program Validation

The program was debugged by running it with the debugging WRITE statements enabled while the game was being played. The printouts of the debugging runs were verified

to assure that correct values were assigned to program variables, that correct program branching logic was followed, and that all program paths were executed at least once. Also, the functions of many of the subroutines (shuffling and dealing the cards, for example) were sufficiently isolated that individual subroutines could be debugged stand-alone rather than in the context of the completed program.

Another aspect of debugging the program has to do with validation of the program's search strategies. The strategies were validated by examining the trace output from the debugging runs. Thus, there could be considered to be two parallel debugging efforts. One was the validation of the program at the instruction-by-instruction level, assuring that each instruction performed the correct function. The other debugging effort was at the broader level of assuring that the program in fact implemented the strategies that it was intended to implement.

Development of New Strategies

In addition to verifying the instruction by instruction logic and the strategies of the program, the debugging runs suggested new strategies to add to the program. It was possible to envision many of the program's strategies during the initial design of the program. However, it was not

realized during the design stage that some of the strategies which were ultimately implemented were in fact required. The end of game and reversible move strategies were particularly difficult to envision. The development process thus consisted of an interaction between the program and the programmer. A series of games would be played until a game was encountered which the program could not decide. A detailed printout of the program's activity in processing the search space for the difficult game would be obtained. Finally, a new strategy would be developed and added to the game based on the printout from the difficult game. The overall process of running the program until it failed, and then developing a new strategy to resolve the failure was repeated until the program could solve most of the games. The implementation of the various E-move and T-move suppression strategies made the most dramatic difference, improving the success rate from about 40% to about 80%.

CHAPTER IV

RESULTS AND FUTURE

CONSIDERATIONS

Necessity for Heuristics

Despite all the strategies that were developed and implemented, there were still games which could not be decided. For some of these games, it was obvious to the human observer that the game could be won. As has already been pointed out, the human observer could take such a game at the point at which the program failed and quickly complete it. It was therefore decided to add some heuristics to the program, and to use the heuristics only after a fully algorithmic approach failed.

Operation of the Program with Heuristics

The operation of the program with heuristics is as follows. The program will play the game up to four times, using different heuristics on the second, third, and fourth attempts. On the first and fourth attempts, the cards are dealt right-to-left and on the second and third attempts they are dealt left-to-right. Permuting the B-columns cannot change whether a game is winnable, but it can change the search time dramatically for winnable games.

The E-Frozen/T-Frozen Heuristic

The first heuristic, called the E-Frozen/T-Frozen heuristic, mimics the suppression strategies. The heuristic goes further than the suppression strategies, and may in fact cause a winnable game to be lost. It nevertheless permits some winnable games to be detected which otherwise would remain undecided. It basically prohibits a card moved into an empty B-column from ever moving to a T-cell, and prohibits a card moved into a T-cell from ever moving into an empty B-column. It is based on the idea that an E-move is almost the same as a T-move, and that an I-move is almost the same as no move at all.

Lack of Progress Heuristic

Several variations of a second heuristic are used on the second, third, and fourth attempts. This heuristic specifically addresses games which the program fails to win, but which the human observer can complete easily. The second heuristic simply forces the program to backtrack when a certain number of consecutive moves are made without making a P-move. The number is varied between 50 and 75 on the second and third attempts.

Speculations on the Undecided Games

The results of running the program indicate that about 75% of all games are known to be winnable and about 15% of

all games are known to be unwinnable, with about 10% of the games in doubt. Results with the E-Frozen/T-Frozen heuristic suggests that about half the games in doubt are unwinnable. That is, about half of the games which are in doubt without the heuristic are determined to be unwinnable with this heuristic. Such a determination, if valid, would raise the number of unwinnable games to 20% and reduce the number of games in doubt to 5%. It appears that the probability that a winnable game would be determined to unwinnable with the E-Frozen/T-Frozen heuristic is quite low. In program testing, there never occurred an example where the moves eliminated by this heuristic were required to win the game. Also, although pathological examples can be constructed where the heuristic eliminates the only path which wins the game, such constructions are extremely difficult.

The remaining 5% of the games which are in doubt also appear to be unwinnable. One particular dichotomy appears between games which are winnable and those which are unwinnable. In all the games which the program decided, winnable games required a search in excess of 100 moves or so deep in the search tree, and unwinnable games always required a search less than 50 moves or so deep into the tree. In earlier versions of the program which did not include heuristics, some undecided games extended more than

100 moves deep, but all such games have subsequently been determined to be winnable with newer versions of the program which include heuristics. (These are the games which the human observer could easily finish). In none of the remaining undecided games did the search extend beyond about 50 moves deep. The program was not at all close to winning these games when the search was abandoned, and it therefore appears that most such games are unwinnable.

The sample size of 300 used for the investigation of the Baker game ($p=0.75$, $q=0.25$) produces a 95% confidence interval of plus or minus 4.9%. Suppose the entire set of undecided games were provisionally assigned to the unwinnable category as suggested by the previous arguments, giving 75% winnable and 25% unwinnable. Even if as many as 50% of the undecided games (5% of the sample) were actually winnable, the error introduced by the provisional assignment would not exceed the statistical sampling error. It appears that the percentage of the undecided games which is actually winnable is quite small, certainly less than 10% (1% of the sample), and perhaps less than 1% (0.1% of the sample). Thus, the total error in the provisional assignment is at most 1%, and it appears that the actual fraction of Baker games which are winnable is not much in excess of 75%.

Future Considerations

For future Baker game research, it is suggested that an attempt might be made to settle the final 10% of the games more definitely. An attempt might be made to refine the E-Frozen/T-Frozen heuristics into rigorous algorithms. Detailed printouts might be obtained of the residual undecidable games in order to search for clues as to additional strategies which might be employed.

During the development of the game, the actual time limit per game was raised from 1 minute to 5 minutes to 10 minutes. Each extension significantly increased the number of games that could be decided. Even with a 10 minute limit, the mean time per game stayed at around 1 minute because many of the games could be decided in 5 or 10 seconds. However, making up to 4 passes per game, a maximum of 40 minutes can be dedicated to one game without ever deciding it. It is suggested that more effort might be placed into making decisions at the point where time limits are exceeded. For instance, if the maximum search level never exceeded 30 or 40, it is likely that additional passes would not win the game. In this case, it would probably be best just to abandon the game entirely if the first move still in the path were a B-move or a T-move such as T1 or T2, but to continue the game rather than starting another

pass if the first move still in the active path were a T-move such as T7 or T8.

If on the other hand, the maximum search level at the 10 minute mark were greater than 100, it is likely that the game is winnable. In this case, it might be best to introduce heuristics designed for completing the last moves of a winnable game, even though the program was still on the first pass.

In any event, it would be quite expensive to obtain results much better than those already obtained, both in terms of analysis and research time, and in terms of machine time. As for machine time, large statistical variations inherent in small samples can be overcome only by sample sizes of at least several thousand, a very expensive computer run for the Baker game. The remaining problems of analysis are quite intractable, also. For instance, in excess of 60,000 nodes can be examined by machine in 10 minutes, and it is such a set of nodes that would have to be analyzed to develop new strategies. It therefore appears that even a large additional effort would not produce results much better than those already obtained. At the most, it is suggested that any additional effort consist of small refinements to the program, and that those refinements concentrate in the area of recognition of whether additional

time should be devoted to the game in progress or whether the game in progress should be abandoned.

LIST OF REFERENCES

LIST OF REFERENCES

- Gardner, M. (1968). Mathematical Games, Scientific American, June.
- Jackson, P.C. (1974). Introduction to Artificial Intelligence. New York: Mason/Charters.
- Knuth, D.E. (1973). Fundamental Algorithms. Reading, Mass.: Addison-Wesley.
- Slagle, J.R. (1971). Artificial Intelligence: The Heuristic Programming Approach. New York: McGraw-Hill.
- Walpole, R.E., and Myers R.H. (1978). Probability and Statistics for Engineers and Scientists, Second Edition. New York: Macmillan.

VITA

Robert Gerald Bryan was born in Knoxville, Tennessee on 18 June 1948. He attended elementary schools in that city and graduated from Knoxville Central High School in June 1966. The following September he entered The University of Tennessee, and in December 1970 he received a Bachelor of Science degree in Mathematics. At the same time, he was commissioned as a Second Lieutenant in the United States Air Force through the The University of Tennessee's ROTC program.

After a tour of duty in the Air Force, he was employed at the University of Tennessee Computing Center in May 1973. In September 1973, he entered the Graduate School of the University of Tennessee, Knoxville, on a part time basis. He received the Master of Science degree with a major in Computer Science in December 1981.

The author is married to the former Cathy Woodhead of Lansing, Michigan. They have two children, Ann and Peter.