THE UNIVERSITY OF CALGARY

Optimal Tunneling:
A Heuristic For Learning Macro Operators

by

Mark L. James

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
APRIL, 1993

© Mark L. James 1993

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-83166-9

Canada

Name MARK JAMES

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science
SUBJECT TERM

0984
SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Nom _____

Dissertation Abstracts International est organisé en catégories de sujets. Veuillez s.v.p. choisir le sujet qui décrit le mieux votre thèse et inscrivez le code numérique approprié dans l'espace réservé ci-dessous.

_____ ☐☐☐ U·M·I

SUJET — CODE DE SUJET

Catégories par sujets

HUMANITÉS ET SCIENCES SOCIALES

COMMUNICATIONS ET LES ARTS

Architecture 0729
Beaux-arts 0357
Bibliothéconomie 0399
Cinéma 0900
Communication verbale 0459
Communications 0708
Danse 0378
Histoire de l'art 0377
Journalisme 0391
Musique 0413
Sciences de l'information 0723
Théâtre 0465

ÉDUCATION

Généralités 515
Administration 0514
Art 0273
Collèges communautaires 0275
Commerce 0688
Économie domestique 0278
Éducation permanente 0516
Éducation préscolaire 0518
Éducation sanitaire 0680
Enseignement agricole 0517
Enseignement bilingue et multiculturel 0282
Enseignement industriel 0521
Enseignement primaire. 0524
Enseignement professionnel 0747
Enseignement religieux 0527
Enseignement secondaire 0533
Enseignement spécial 0529
Enseignement supérieur 0745
Évaluation 0288
Finances 0277
Formation des enseignants 0530
Histoire de l'éducation 0520
Langues et littérature 0279
Lecture 0535
Mathématiques 0280
Musique 0522
Orientation et consultation 0519
Philosophie de l'éducation 0998
Physique 0523
Programmes d'études et enseignement 0727
Psychologie 0525
Sciences 0714
Sciences sociales 0534
Sociologie de l'éducation 0340
Technologie 0710

LANGUE, LITTÉRATURE ET LINGUISTIQUE

Langues
- Généralités 0679
- Anciennes 0289
- Linguistique 0290
- Modernes 0291

Littérature
- Généralités 0401
- Anciennes 0294
- Comparée 0295
- Médiévale 0297
- Moderne 0298
- Africaine 0316
- Américaine 0591
- Anglaise 0593
- Asiatique 0305
- Canadienne (Anglaise) 0352
- Canadienne (Française) 0355
- Germanique 0311
- Latino-américaine 0312
- Moyen-orientale 0315
- Romane 0313
- Slave et est-européenne 0314

PHILOSOPHIE, RELIGION ET THÉOLOGIE

Philosophie 0422
Religion
- Généralités 0318
- Clergé 0319
- Études bibliques 0321
- Histoire des religions 0320
- Philosophie de la religion 0322

Théologie 0469

SCIENCES SOCIALES

Anthropologie
- Archéologie 0324
- Culturelle 0326
- Physique 0327

Droit 0398
Économie
- Généralités 0501
- Commerce-Affaires 0505
- Économie agricole 0503
- Économie du travail 0510
- Finances 0508
- Histoire 0509
- Théorie 0511

Études américaines 0323
Études canadiennes 0385
Études féministes 0453
Folklore 0358
Géographie 0366
Gérontologie 0351
Gestion des affaires
- Généralités 0310
- Administration 0454
- Banques 0770
- Comptabilité 0272
- Marketing 0338

Histoire
- Histoire générale 0578
- Ancienne 0579
- Médiévale 0581
- Moderne 0582
- Histoire des noirs 0328
- Africaine 0331
- Canadienne 0334
- États-Unis 0337
- Européenne 0335
- Moyen-orientale 0333
- Latino-américaine 0336
- Asie, Australie et Océanie 0332

Histoire des sciences 0585
Loisirs 0814
Planification urbaine et régionale 0999
Science politique
- Généralités 0615
- Administration publique 0617
- Droit et relations internationales 0616

Sociologie
- Généralités 0626
- Aide et bien-être social 0630
- Criminologie et établissements pénitentiaires 0627
- Démographie 0938
- Études de l' individu et de la famille 0628
- Études des relations interethniques et des relations raciales 0631
- Structure et développement social 0700
- Théorie et méthodes. 0344
- Travail et relations industrielles 0629

Transports 0709
Travail social 0452

SCIENCES ET INGÉNIERIE

SCIENCES BIOLOGIQUES

Agriculture
- Généralités 0473
- Agronomie. 0285
- Alimentation et technologie alimentaire 0359
- Culture 0479
- Élevage et alimentation 0475
- Exploitation des pâturages 0777
- Pathologie animale 0476
- Pathologie végétale 0480
- Physiologie végétale 0817
- Sylviculture et faune 0478
- Technologie du bois 0746

Biologie
- Généralités 0306
- Anatomie 0287
- Biologie (Statistiques) 0308
- Biologie moléculaire 0307
- Botanique 0309
- Cellule 0379
- Écologie 0329
- Entomologie 0353
- Génétique 0369
- Limnologie 0793
- Microbiologie 0410
- Neurologie 0317
- Océanographie 0416
- Physiologie 0433
- Radiation 0821
- Science vétérinaire 0778
- Zoologie 0472

Biophysique
- Généralités 0786
- Medicale 0760

SCIENCES DE LA TERRE

Biogéochimie 0425
Géochimie 0996
Géodésie 0370
Géographie physique 0368
Géologie 0372
Géophysique 0373
Hydrologie 0388
Minéralogie 0411
Océanographie physique 0415
Paléobotanique 0345
Paléoécologie 0426
Paléontologie 0418
Paléozoologie 0985
Palynologie 0427

SCIENCES DE LA SANTÉ ET DE L'ENVIRONNEMENT

Économie domestique 0386
Sciences de l'environnement 0768
Sciences de la santé
- Généralités 0566
- Administration des hôpitaux 0769
- Alimentation et nutrition 0570
- Audiologie 0300
- Chimiothérapie 0992
- Dentisterie 0567
- Développement humain 0758
- Enseignement 0350
- Immunologie 0982
- Loisirs 0575
- Médecine du travail et thérapie 0354
- Médecine et chirurgie 0564
- Obstétrique et gynécologie 0380
- Ophtalmologie 0381
- Orthophonie 0460
- Pathologie 0571
- Pharmacie 0572
- Pharmacologie 0419
- Physiothérapie 0382
- Radiologie 0574
- Santé mentale 0347
- Santé publique 0573
- Soins infirmiers 0569
- Toxicologie 0383

SCIENCES PHYSIQUES

Sciences Pures

Chimie
- Généralités 0485
- Biochimie 487
- Chimie agricole 0749
- Chimie analytique 0486
- Chimie minérale 0488
- Chimie nucléaire 0738
- Chimie organique 0490
- Chimie pharmaceutique 0491
- Physique 0494
- PolymÇres 0495
- Radiation 0754

Mathématiques 0405
Physique
- Généralités 0605
- Acoustique 0986
- Astronomie et astrophysique 0606
- Électronique et électricité 0607
- Fluides et plasma 0759
- Météorologie 0608
- Optique 0752
- Particules (Physique nucléaire) 0798
- Physique atomique 0748
- Physique de l'état solide 0611
- Physique moléculaire 0609
- Physique nucléaire 0610
- Radiation 0756

Statistiques 0463

Sciences Appliqués Et Technologie

Informatique 0984
Ingénierie
- Généralités 0537
- Agricole 0539
- Automobile 0540
- Biomédicale 0541
- Chaleur et ther modynamique 0348
- Conditionnement (Emballage) 0549
- Génie aérospatial 0538
- Génie chimique 0542
- Génie civil 0543
- Génie électronique et électrique 0544
- Génie industriel 0546
- Génie mécanique 0548
- Génie nucléaire 0552
- Ingénierie des systèmes 0790
- Mécanique navale 0547
- Métallurgie 0743
- Science des matériaux 0794
- Technique du pétrole 0765
- Technique minière 0551
- Techniques sanitaires et municipales 0554
- Technologie hydraulique 0545

Mécanique appliquée 0346
Géotechnologie 0428
Matières plastiques (Technologie) 0795
Recherche opérationnelle 0796
Textiles et tissus (Technologie) 0794

PSYCHOLOGIE

Généralités 0621
Personnalité 0625
Psychobiologie 0349
Psychologie clinique 0622
Psychologie du comportement 0384
Psychologie du développement 0620
Psychologie expérimentale 0623
Psychologie industrielle 0624
Psychologie physiologique 0989
Psychologie sociale 0451
Psychométrie 0632

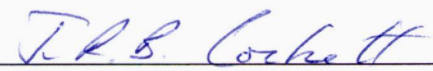
THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "Optimal Tunneling: A Heuristic For Learning Macro Operators" submitted by Mark L. James in partial fulfillment of the requirements for the degree of Master of Science.

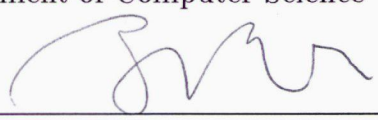
Supervisor, Bruce MacDonald,

Department of Computer Science



Robin Cockett,

Department of Computer Science



Jun Gu,

Department of Electrical Engineering

Date May 7th 1993

Abstract

This thesis discusses methods for efficiently solving puzzles by heuristic (best first) search, concentrating on techniques for learning *macro operators*. The thesis focuses on the *Optimal Tunneling* heuristic for learning macros, which produces shorter, more useful macros than previous techniques. The Optimal Tunneling heuristic proposes macros from a state at a minimum to the next state with the same heuristic value. A water pouring analogy is used to show that the "horizontal tunnels" learned by the Optimal Tunneling heuristic are more appropriate than "aqueducts", or "sloping tunnels". Macros learned by the Optimal Tunneling heuristic avoid expensive search against the heuristic function, and give the best modification of the search space to make the heuristic function more accurate. The thesis discusses the high sensitivity of best first search to arbitrary design choices in selecting states with the same heuristic value. To minimize the effects of this problem, all testing was performed with a single problem solver. Comparative tests in the domains of Peg Solitaire, Tile Sliding and SOKOBAN show that the Optimal Tunneling heuristic results in a clear improvement over previous techniques for both single trials, and test sequences with cumulative learning across trials.

Acknowledgements

My deepest gratitude to my supervisor, Bruce MacDonald, who was always available with advice, feedback, and support.

My thanks to Glen Iba who provided helpful advice and interesting commentary in the early stages of my thesis.

I was fortunate to have many proofreaders: Eric Schenk, Carol Wang, Thomas Orth, Greg James, Natascha Schuler, Theo Deraadt and my parents, Alan and Margaret James. All provided valuable feedback during the finishing stages of my thesis.

Dedication

This thesis is dedicated to my late grandfather, Ralph James.

Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

> *The "Fifteen" puzzle for the last few weeks has been prominently before the American public, and may safely be said to have engaged the attention of nine out of ten persons of both sexes and of all ages and conditions of the community* — W.W. Johnson (1879)

Puzzles have facinated people for hundreds, if not thousands of years. For example, there is stong evidence that the game of Peg Solitaire was played as early as 1697, and was probably played long before that [4]. Researchers in Artificial Intelligence and Machine Learning often use puzzles as example domains for problem solving systems. This thesis investigates methods for efficiently solving puzzles such as the "Fifteen" puzzle, Peg Solitaire, and SOKOBAN.

1.1. Macro-operators

The classical technique for solving a puzzle by computer involves searching for a sequence of *operators*. A typical system will use *heuristic search*, in which a heuristic function estimates the distance to the goal from a given state [3]. This allows the system to examine only the most promising states.

Often it is useful to group subsequences of operators into chunks to form composite operators. These subsequences are called *macro-operators* or *macros*, and can be used in the same way as primitive operators [12, 15]. Macros can lead to faster solutions

that are obtained by less effort in the search. However, macro generation increases the size of the operator set and can therefore result in a more expensive search. These two effects must be managed by the macro generator and filtering, so that the overall result is computationally beneficial.

Some systems generate only macros that skip the expensive parts of a search, where the heuristic function is increasing. Iba's [13] Minimum to Minimum heuristic is an example. The Minimum to Minimum heuristic proposes macros from a state at a minimum to the next state at a minimum. However, it creates macros that are longer than necessary, which may mean they are less useful in some domains. This thesis presents the *Optimal Tunneling* heuristic, which proposes macros from a state at a minimum to the next state with the same heuristic value.

The thesis is organized as follows. Chapter 2 gives an overview of a broad range of problem-solving techniques that can be grouped under the general heading of *search*. These techniques can be used for a wide range of problems, including the solution to puzzles. Chapter 3 describes Iba's MACLEARN system [13]. MACLEARN improves on the basic search techniques presented in Chapter 2 by learning macro-operators to speed up problem solving. These macros can also be applied to more complex problems in similar domains, allowing MACLEARN to use information learned in simple training problems to solve more complex problems.

MACLEARN uses a Minimum to Minimum heuristic to learn macro operators. This heuristic works well for domains where minima are close together. Chapter 4 gives the motivation for and describes the Optimal Tunneling heuristic, which generates shorter macros than Minimum to Minimum, allowing it to work well in domains where minima are farther apart. A water pouring analogy is examined, showing that Optimal Tunneling should result in an improvement.

Chapter 5 describes and interprets comparative tests on the performance of the two macro learning heuristics. Tests on the Peg Solitaire and Tile Sliding domains show that Optimal Tunneling works better even on domains where Minimum to Minimum works well. Tests on the SOKOBAN domain show that when minima are farther apart, Optimal Tunneling can solve complex problems in much less time than the Minimum to Minimum heuristic.

Chapter 6 contains concluding remarks and discusses future research. It examines some limitations of Iba's learning model and suggests some possible improvements.

1.2. Contribution

Iba's MACLEARN is a useful problem solving system. It can use macro operators to solve several complex problems that cannot be solved with heuristic search alone. I have reimplemented MACLEARN and analyzed its performance for the domains of Peg Solitaire, Tile Sliding, and SOKOBAN. An analysis of the SOKOBAN domain suggested that the Minimum to Minimum heuristic would work poorly for SOKOBAN since the minima are far apart.

This thesis presents the Optimal Tunneling heuristic, an improvement to the Minimum to Minimum heuristic. A water pouring analogy suggests that the progress of a best first search is slowed by "valleys" along the solution path. Macro operators act like tunnels, allowing the water to drain from valleys without forming "lakes." This analogy is used to explain why the macros proposed by the Optimal Tunneling heuristic result in more efficient search than those proposed by the Minimum to Minimum heuristic, or any similar heuristic.

The thesis discusses the high sensitivity of best first search to arbitrary design choices in selecting states with the same heuristic value. To minimize the effects of this problem all testing was performed with a single problem solver. The Optimal

Tunneling heuristic was compared to the Minimum to Minimum heuristic using the learning model described in Section 3.1. Comparative tests in the domains of Peg Solitaire, Tile Sliding and SOKOBAN show that the Optimal Tunneling heuristic results in a clear improvement over previous techniques for both single trials, and test sequences with cumulative learning across trials.

These ideas and results presented in this thesis will allow researchers to solve complex puzzles more efficiently. The water pouring analogy can be extended to domains other than puzzle solving, and provides an easy method of understanding the effects of macro operators on best first search. The Optimal Tunneling heuristic may prove useful in a variety of other problem solving domains such as symbolic integration and robot task planning.

CHAPTER 2

Search

This chapter discusses a broad range of problem-solving techniques that can be grouped under the general heading of *search.* Barr and Feigenbaum [2] describe search as a name for a large body of core ideas that deal with deduction, inference, planning, common-sense reasoning, theorem proving, and related processes. Applications of these ideas are found in programs for natural language understanding, information retrieval, automatic programming, robotics, scene analysis, game playing, expert systems, machine learning, mathematical theorem proving, and puzzle solving.

Section 2.1 discusses problem representations that form the basis of search techniques. Section 2.2 examines common search methods. Section 2.3 discusses several systems that use *macro operators* to speed up search.

2.1. State Space Representation

A *state space* is a convenient way to represent a search problem. The *state space representation* of a problem consists of *states*, which represent the configuration of a problem at a given time, and *operators*, which transform the problem from one state to another. A common example of state space representation is the Eight puzzle [4] (Figure 2.1). The Eight puzzle consists of eight tiles in a square grid with a space in place of the ninth tile. A tile may be moved either horizontally or vertically by sliding it into the empty square. The problem is to find a sequence of operators that

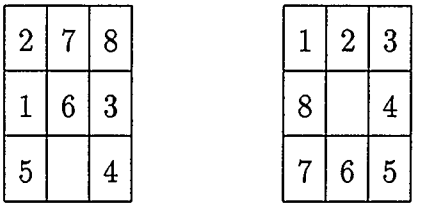


FIGURE 2.1. Two States Of The Eight Puzzle

will transform a given state, such as that in Figure 2.1(a) into a goal state, such as that in Figure 2.1(b). The states may be represented as a 3×3 array. The operators can be defined as moving tiles into the blank square, giving four operators:

UP Move the tile below the blank up one square.

DOWN Move the tile above the blank down one square.

LEFT Move the tile to the right of the blank left one square.

RIGHT Move the tile to the left of the blank right one square.

The complete specification of a state space problem consists of O, the description of the operators, S, a set of one or more initial states, and G, a predicate defining goal states. A solution to the problem is a sequence of operators that transforms an initial state into a goal state. Section 2.2 describes several algorithms for solving state space problems.

2.2. Search Methods

A search algorithm for a state space problem attempts to find a sequence of operators that will transform an initial state into a goal state. *Expanding* a state is the basic operation used in the search algorithms presented in this section. A state is expanded by applying all applicable operators to it to generate a list of new states. The basic search algorithm is shown in Algorithm 2.1.

ALGORITHM 2.1 (BASIC SEARCH).

(1) *Put the start state in a list of unexpanded states called OPEN. If the start state is a goal state, a solution has been found.*
(2) *If OPEN is empty, no solution exists.*
(3) *Remove the first state, i, from OPEN, and place it in a list of expanded states called CLOSED.*
(4) *Expand state i. For every successor j of i: If j is neither in OPEN nor CLOSED, add j to OPEN. The order that states are inserted into OPEN depends on the search algorithm used. Attach a pointer from j back to its predecessor i (to trace a solution path once a goal is found).*
(5) *If any of the successors of i is a goal state, a solution has been found, otherwise, go to (2).*

ALGORITHM 2.2 (BREADTH FIRST SEARCH).
(1) *Use the basic search algorithm (Algorithm 2.1), placing newly generated states at the end of the OPEN list.*

2.2.1. Brute Force Methods. The simplest method for solving a search problem is a brute force search. This technique needs no domain knowledge. For this reason, it is often called a *blind search.* The three types of brute force search discussed in this section are based on Algorithm 2.1, and differ only in the order that they insert new states into the OPEN list in step (4).

2.2.1.1. *Breadth First Search.* The Breadth First Search (Algorithm 2.2) expands those states that are closest to the start state. It examines all possible sequences of n operators, then all possible sequences of $n+1$ operators, and so on. Since it always examines shorter paths first, it is guaranteed to find the shortest solution if one exists. The biggest disadvantage of the Breadth First Search is its space complexity. If there are b applicable operators from each state in a search space, and the length of the shortest solution is l, then the Breadth First Search needs $O(b^l)$ time, and $O(b^{l-1})$ memory. Iterative Deepening (Section 2.2.1.3) simulates a Breadth First Search, but uses only $O(l)$ space, at some cost to the time required.

2.2.1.2. *Depth First Search.* As the name suggests, Depth First Search (Algorithm 2.3) expands the deepest state first. It expands a single path through the search space until the last state of that path has no successors, and only then does it consider a different path. Unlike the Breadth First Search, Depth First Search is not guaranteed to find the shortest solution. Since only the current path through the search space is stored, Depth First Search requires $O(l)$ space, where l is the length of the longest path considered. Like Breadth First Search, Depth First Search must, in the worst case, examine every state in the search space. If each state has b successors,

ALGORITHM 2.3 (DEPTH FIRST SEARCH).
(1) *Use the basic search algorithm (Algorithm 2.1), placing newly generated states at the beginning of the OPEN list.*

ALGORITHM 2.4 (ITERATIVE DEEPENING).
(1) *Set MAXDEPTH, the Maximum Depth for Depth First Search, to 1.*
(2) *Call Depth First Search, and record the longest path explored.*
(3) *If a solution was found, then terminate.*
(4) *If the longest path is less than MAXDEPTH, then no solution exists. Terminate.*
(5) *Increment the Maximum Depth.*
(6) *Go to (2).*

Depth First Search needs $O(b^l)$ time. If the search space is infinite, the Depth First Search may never terminate. To prevent this, a maximum depth is usually specified for the search. Any states deeper than this maximum depth are treated as if they had no successors.

2.2.1.3. *Iterative Deepening.* Iterative Deepening (Algorithm 2.4) solves the memory problems of the Breadth First Search at some cost to the time required. The algorithm works by using Depth First Search to examine all paths of length 1, then all paths of length 2, and so on. Since it examines all paths of length n before examining any longer paths, this algorithm is guaranteed to find the shortest solution. If the shortest path found is of length l, Iterative Deepening needs only $O(l)$ memory to find the path. The time complexity of Iterative Deepening is $O(\sum_{i=1}^{l} b^i) = O(2b^l) = O(b^l)$. The time "overhead" of this method is usually minor compared to the memory requirements for Breadth First Search.

2.2.2. Heuristic Search. In a blind search, the number of states examined before a solution is found is likely to be prohibitively large. Since no knowledge of the problem domain is used, it is unlikely that any but the simplest of problems can be

ALGORITHM 2.5 (HEURISTIC SEARCH).

(1) *Use the basic search algorithm (Algorithm 2.1)*
(2) *For each newly generated state j:*
    (a) *Calculate $f^*(j)$.*
    (b) *If j is neither in OPEN nor CLOSED, add j to OPEN, sorted into ascending order by f^* value.*
    (c) *If j is in either OPEN or CLOSED, compare the f^* value just calculated for j with the value previously associated with the state. If the new value is lower then*
        (i) *Substitute it for the old value.*
        (ii) *Point j back to i instead of its previous predecessor.*
        (iii) *If state j was on the CLOSED list, move it back to OPEN.*

solved before running out of resources. *Heuristic search* uses knowledge about the domain to choose the most promising state to investigate.[1].

Heuristic search, also known as an *ordered* or *best first* search always selects the most promising state to expand. The "promise" of a state is given by an *evaluation function.* The evaluation function f^* is defined so that more promising states have smaller values of f^*. For a state space problem, the promise of a state is often defined as an estimate of its distance from a goal state.

Given the evaluation function, or *heuristic function*, f^*, Algorithm 2.5 tries to reduce the number of states expanded by blind search. The success of this algorithm depends on the choice of f^*.

2.3. Macro Operators

2.3.1. STRIPS. STRIPS [8] is a general problem solver based on a combination of *means-ends analysis* [16] and mathematical theorem proving. STRIPS attempts to find a sequence of operators, or *plan*, that will transform a given start state into a state that satisfies a goal. A later addition to STRIPS allows it to generalize its

[1]Barr and Feigenbaum [2] describe the history of the terms "heuristic" and "heuristic search".

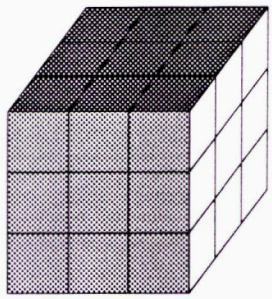


FIGURE 2.2. Rubik's Cube

plans and use them again for other problems [7]. Such a generalized plan is called a *MACROP*. If a MACROP can be used as part of a new plan, the time needed to find the plan may be considerably reduced.

One disadvantage of the learning technique used by STRIPS is that it learns new MACROPs without considering their utility. As new problems are solved, the set of MACROPs grows larger and larger until, eventually, the time spent checking each MACROP is greater than the time saved by creating them.

2.3.2. Solving Subgoals. Heuristic search uses an evaluation function to estimate the distance from a given state to the goal state. This technique is much more efficient than brute-force search, but for some problems, such as Rubik's Cube (Figure 2.2), it is difficult to find a heuristic evaluation function that increases monotonically towards the goal.

The standard Rubik's Cube [9] consists of 26 subcubes arranged as a $3 \times 3 \times 3$ cube. The visible faces of these subcubes are called *facelets*. The goal is to arrange these subcubes or *cubies* so that the faclets on each cube face are the same color. This can be done by rotating the faces of the cube (Figure 2.3). Two similar puzzles are the $2 \times 2 \times 2$ "Rubik's Pocket Cube" and the $4 \times 4 \times 4$ "Rubik's Revenge." Korf [15] examined several possible heuristic evaluation functions for the Rubik's Cube family

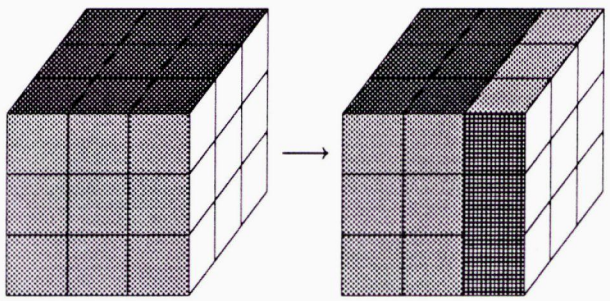


FIGURE 2.3. The Effect Of Rotating A Cube Face

of puzzles by enumerating all of the states of the Rubik's Pocket Cube ($2 \times 2 \times 2$), and found that none of them were useful for solving the $2 \times 2 \times 2$ cube. It seems unlikely that any useful evaluation function can be computed easily, which suggests that Rubik's Cube puzzles cannot be solved with heuristic search.

Many problems can be easily expressed as a composition of sub-problems. For example, Rubik's Cube can be expressed as "Get each cubie to the correct position and orientation." This suggests setting up a sequence of subgoals and solving them one at a time. The General Problem Solver (GPS) [16] implements means-ends analysis along with other problem-solving techniques. It is applicable if there exist a set of subgoals and an ordering of these goals such that, once a goal is satisfied, it need never be violated to solve the remaining subgoals [6]. A set of subgoals with this property is called *serializable.*

Unfortunately, Rubik's Cube does not satisfy this condition. Once some cubies have been placed in their goal positions, they must be moved to solve the other cubies. What is needed here is to find a way to solve a subgoal in such a way that any previously solved subgoals, although they may move temporarily, are left in the proper state.

2.3.3. The Macro Problem Solver (MPS). Korf's Macro Problem Solver (MPS) [15] solves problems such as Rubik's cube and the Eight puzzle in this way by creating a macro table.

2.3.3.1. *The Macro Table.* The goal of this system is to build a macro table (Table 2.1). Each entry in the table gives a macro to solve a subgoal so that any previously solved subgoals, although they may be destroyed temporarily, are eventually restored.

Figure 2.4 shows how the macros in Table 2.1 are used to solve the Eight puzzle. State (a) is the initial state of the puzzle we want to solve. First we find the blank,

	0	1	2	3	4	5	6
0							
1	LU						
2	U	RDLU					
3	UR	DLURRDLU	DLUR				
4	R	LDRURDLU	LDRU	RDLLURDRUL			
5	DR	LURDLDRURDLU	LURDLDRU	LDRULURDDLUR	LURD		
6	D	URDLDRUL	ULDDRU	LDRUULDRDLUR	ULDR	RDLLUURDLDRRUL	
7	LD	RULDDRUL	URDLULDDRU	RULDRDLULDRRUL	URDLULDR	ULDRURDLLURD	URDL
8	L	DRUL	RULLDDRU	RDLULDRRUL	RULLDR	ULDRRULDLURD	RULD

TABLE 2.1. Macro Table For The Eight Puzzle

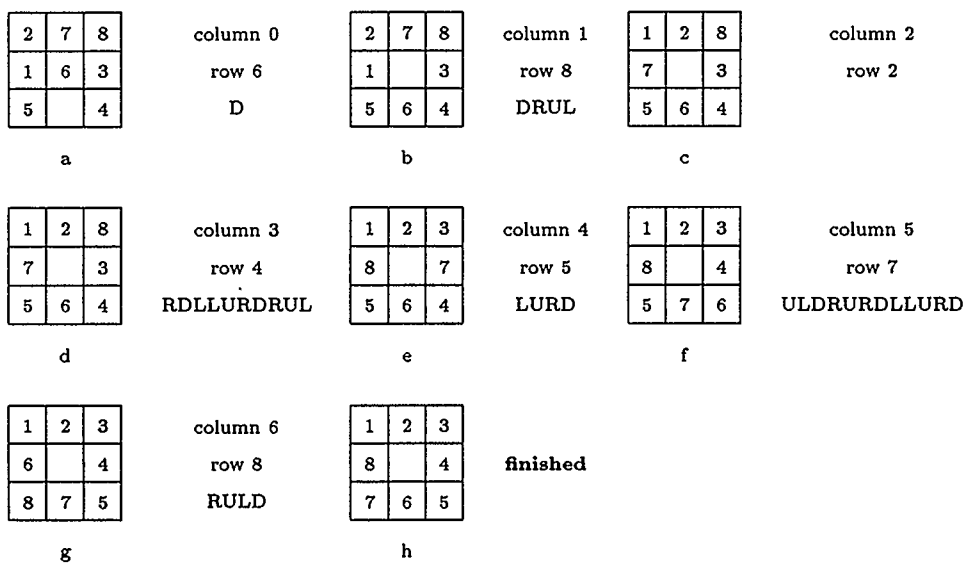


FIGURE 2.4. Sample solution of the Eight Puzzle by MPS

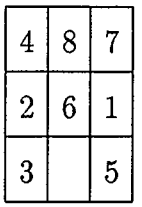


FIGURE 2.5. An Arbitrary State Of The Eight Puzzle, Its State-Vector Representation Is $(6, 4, 8, 7, 1, 5, 0, 3, 2)$

which is in the position where the 6 belongs. Column 0 contains the macros for the blank, and the macro in row 6 is "D." This macro moves the blank to its goal position in the center. In state (b) tile 1 is in position 8. The macro in row 1 column 8 is "DRUL." This macro moves tile 1 to its goal position leaving the blank in its goal position. During the application of the macro, the blank is moved out of its goal position, but it is always restored to the goal position by the end of the macro. In state (c) tile 2 is already in position 2, so no macro is applied. Tile 3 is in position 4 in state (d), so the macro from column 3, row 4 ("RDLLURDRUL") is applied. This continues until each tile is placed in its goal position. Using the macro table involves no search, and is an efficient solution technique.

2.3.3.2. *The Basic Algorithm.* In order for the technique in Section 2.3.3.1 to work, the Macro Problem Solver must first build the macro table. The basic algorithm is to fill in the entries in the table by searching the space of possible macro operators. Each macro generated is inserted into the table in its correct slot, unless a shorter macro has already filled that slot.

A search through the space of possible macros can be accomplished by an iterative deepening search from the goal. MPS represents each possible state of the puzzle as a vector of state variables. For a given state, each variable corresponds to a piece of the puzzle, and the value of each variable corresponds to the position of that piece.

For example, the state variables for the Eight puzzle correspond to the blank, and the eight sliding tiles. The value of the variable corresponds to the position of the tile in the goal state. Figure 2.5 shows an arbitrary state of the Eight puzzle, and its state vector representation. The blank is in the position corresponding to the 6 tile, so the 0th element of the vector is 6. The 1 tile is in position 4, so the 1st element of the vector is 4, and so on. Note that 0 is used to represent the blank. Although different representations exist, some of which may seem more intuitive, MPS is dependent on the representation, and in general, it will not work with other representations.

Given that we can find macros with an iterative deepening search from the goal, we need to be able to decide where the macros fit in the table. If a macro belongs in column n, row m of the table, then that means the macro moves a tile from position m to position n, leaving all the pieces from positions $0 \ldots (n-1)$ unchanged. Now consider applying the inverse of that macro to the goal state. It will take the tile from position n and move it to position m while leaving the tiles from positions $0 \ldots (n-1)$ unchanged. The state vector for the state obtained by applying the inverse of this macro to the goal state will look like $(0, 1, \ldots, (n-1), m, \ldots)$. The first n values of the state vector will correspond to the goal values. A macro that when applied to the goal leaves the first n state variables unchanged is said to have an *invariance* of n. Since the invariance of the macro gives the longest sequence of variables that are unchanged by the macro, the invariance of the macro tells us which column the macro belongs in. If a macro has invariance n, then so does its inverse; since the macro does not change the values of the first n state variables, neither will its inverse.

In addition, the value of the nth state variable tells us which row the macro's inverse belongs in. So for each state found during the search from the goal, we find the invariance of the macro and look at the value of first state variable that differs

from the goal. If the invariance of the macro is n, and the value of the first state variable that differs from the goal is m, then we place the inverse of the macro in column n, position m in the table. Since the search finds all the length one macros, then all the length two macros, and so on, we will always find the shortest macros first. Thus we can simply put a macro in the table if there is no macro already there. If the longest macro in the completed table is of length n, this technique will find all the macros with a search to depth n.

2.3.3.3. *Partial Match, Bi-directional Search.* Consider two macros that, when applied to the goal state, map the ith state variable to the same value. If we apply the first macro followed by the inverse of the second macro to the goal, the effect will be to move the ith state variable and then to move it back its goal value. So any two macros that when applied to the goal generate an identical sequence of i state variables can be composed to generate a new macro of invariance i. By storing the state vectors of each macro when applied to the goal, it is possible to generate macros of length $2i$ by composing two macros of length i whose first n state variables are equal (but not necessarily equal to the goal value).

The above technique is based on the bi-directional search described by Pohl [17]. The major differences are that we use only one search from the goal, and that only the first n state variables must match. This allows us to find macros of length d with a search to depth $d/2$, which is a tremendous computational advantage since it reduces the cost of the search from b^d to $b^{d/2}$ where b is the average branching factor. However, this technique does not gain us much if each new state must be compared with each existing state. When this is the case, the bi-directional search takes the same amount of time as an ordinary search, with comparisons taking most of the

time. Fortunately, it is possible to hash the states based on the values of the state variables.

2.3.3.4. *Macro Composition.* When we compose two macros of invariance i, the resulting macro will be a different macro having invariance i. If, when applied to the goal state, the $(i+1)$th state variables are equal, then if we compose one macro with the inverse of the other, we will get a macro with invariance of at least $i+1$. Korf takes advantage of this by first letting the algorithm above run until memory is exhausted, and then composing the macros with the highest invariance to fill the empty slots in the table [14]. This technique has the advantage that it can find high invariance macros with little computation, however, it is not guaranteed to find the shortest macros.

2.4. Summary

Heuristic search can solve problems well with an accurate heuristic function. Some problems such as Rubik's Cube do not have an accurate heuristic function that is easily computable. Korf's MPS (Section 2.3.3) can solve problems such as Rubik's Cube without a heuristic function by building a macro table.

MPS can be used to solve any problem that is *serially decomposable.* Peg Solitaire is an example of a problem that is not serially decomposable, and so cannot be solved by MPS. Another disadvantage of MPS is that information from one domain cannot be transferred to a similar domain. Iba's MACLEARN [13] addresses these problems, and is described in Chapter 3.

CHAPTER 3

Minimum to Minimum

Iba [13] describes a heuristic approach to the discovery of useful macro operators. His system, MACLEARN, learns new macros during problem solving so that they can be used immediately. By learning macros on simple training problems, the system is able to solve much more difficult problems. This chapter describes Iba's MACLEARN system. Section 3.1 describes his general framework for learning macros. Section 3.2 describes details of the system itself. Section 3.3 discusses the results of Iba's tests. Chapter 4 analyses MACLEARN in more detail and introduces an improvement.

3.1. Iba's Learning Model

Iba [13] gives a general framework for learning macros in a wide variety of domains. The learning model is based on the components shown in Figure 3.1. The Performance Element executes some form of search over the operator set. As it finds new states of the puzzle, it passes that information on to the Macro Proposer which proposes new macros at certain stages of the search. The Static Filter decides if a new macro is likely to be useful, then adds it to the operator set. The Dynamic Filter is invoked after a training session to remove macros that were not useful.

The model is flexible, allowing for both in-trial learning, and cumulative learning across similar problems.

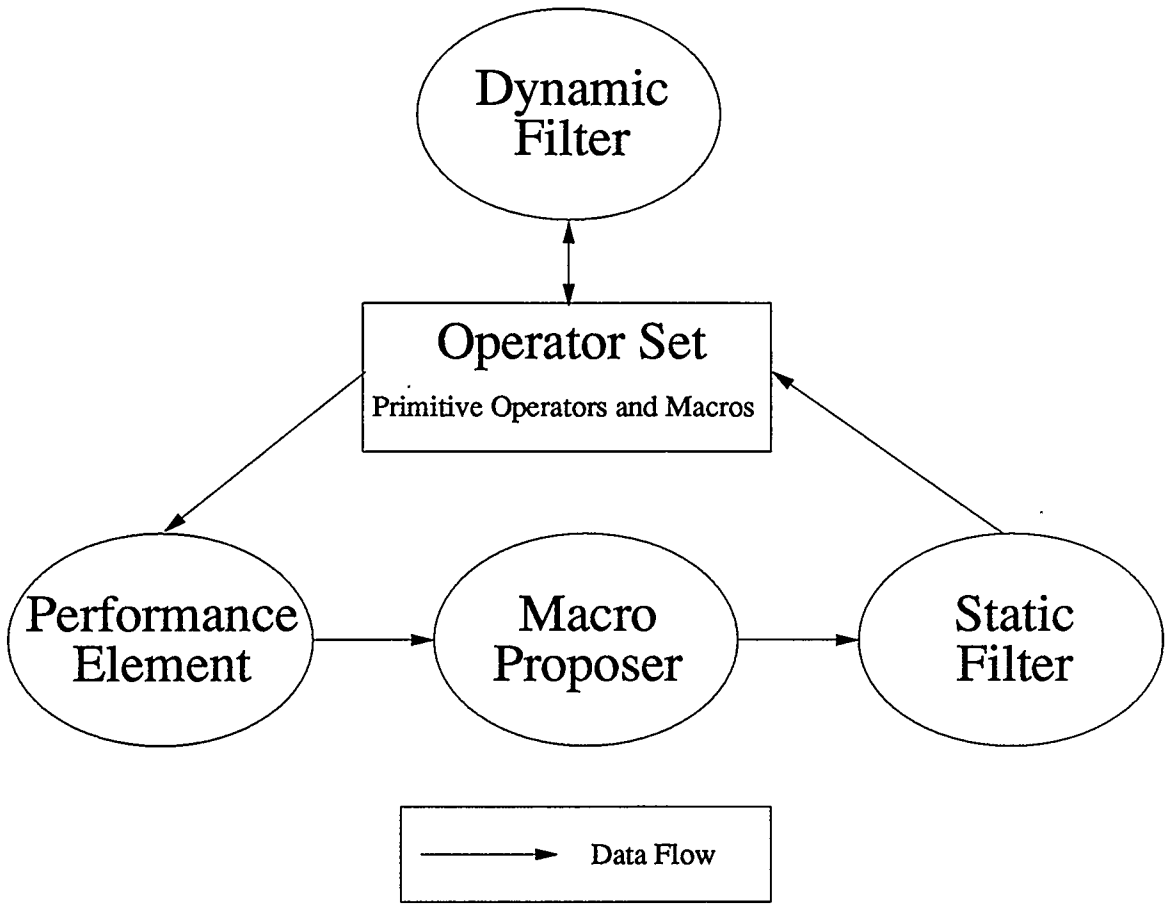


FIGURE 3.1. Iba's Learning Model

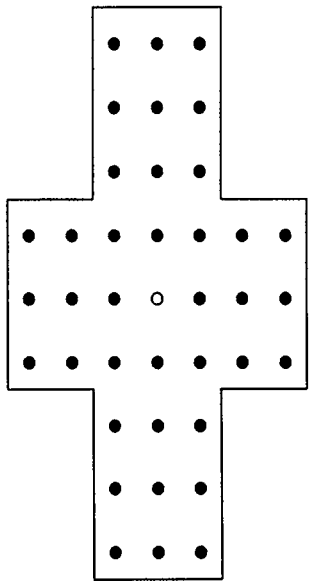


FIGURE 3.2. The Hi-Q Puzzle: An Example Of Peg Solitaire

3.2. Maclearn

Iba's system based on the framework described above is called MACLEARN. Each of the system's components are described in detail below.

3.2.1. Operator Set. MACLEARN operators are represented as pairs of rectangular arrays, that match when reflected or when rotated by 0°, 90°, 180°, or 270°. The first element of the pair represents the preconditions of the operator; the second represents the effects. For example, in the Peg Solitaire domain [4] (Figure 3.2), the only basic operator is

• • o → o o •

Macros are represented in the same way, with the addition of "." which is a *Don't Care* symbol. An example is shown in Figure 3.3.

The choice of representation is important, and the beauty of this one is that it allows macros to be treated in exactly the same way as it treats primitive operators.

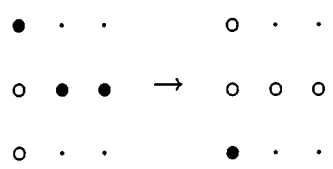


FIGURE 3.3. An Example Of A Macro Operator For Peg Solitaire

ALGORITHM 3.1 (COMPOSE MACRO).

(1) *Find the smallest rectangle that includes all of the locations used by any operator in the sequence being composed.*
(2) *Create an array with the dimensions of this window, and fill it with* Don't Care *symbols. For each operator in the sequence being composed do (in the order that they occur in the sequence)*
    (a) *Change the position of the operator so that it is relative to the rectangle being used rather than the whole board.*
    (b) *Copy the* after *array of the operator into the rectangle being used.* Don't Care *symbols are not copied.*
(3) *Copy the current rectangle to a new* after *array.*
(4) *For each operator in the sequence being composed (in the reverse order that they occur in the sequence)*
    (a) *Change the position of the operator so that it is relative to the rectangle being used rather than the whole board.*
    (b) *Copy the* before *array of the operator into the rectangle being used. Again,* Don't Care *symbols are not copied.*
(5) *Copy the current rectangle to a new* before *array.*

In order for a macro to be useful, it must be generalized so that it can be used in different parts of the search. In MACLEARN, the macros are generalized implicitly by the representation, which allows them to be used anywhere on the puzzle, and at any orientation. There is no explicit generalization step; the generalization comes free with the representation.

3.2.2. Macro Proposer. The Macro Proposer watches the search develop and, based on heuristic rules I will describe shortly, decides when to learn new macros. Once that decision is made, a macro is composed from the chosen operator sequence. The before and after patterns of the operator are created as in Algorithm 3.1. Figure 3.4 shows several steps in the solution to a Peg Solitaire problem. The primitive operators used in this sequence are shown in Figure 3.5. The macro operator in Figure 3.6 is learned by composing these primitive operators.

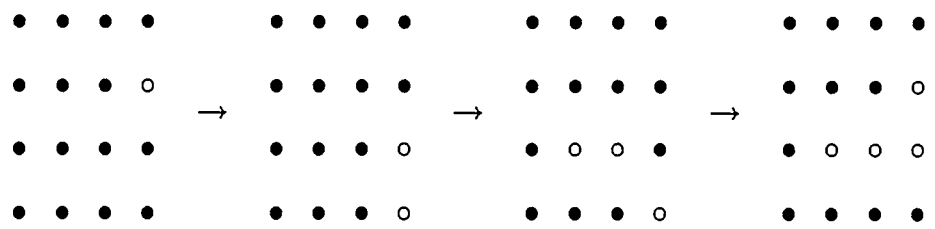


FIGURE 3.4. An Example Operator Sequence For Macro Composition

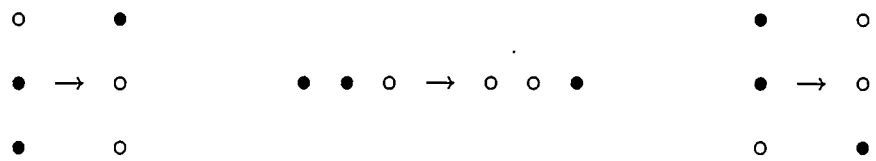


FIGURE 3.5. Operators Used In Figure 3.4

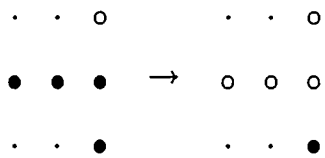


FIGURE 3.6. Macro Operator

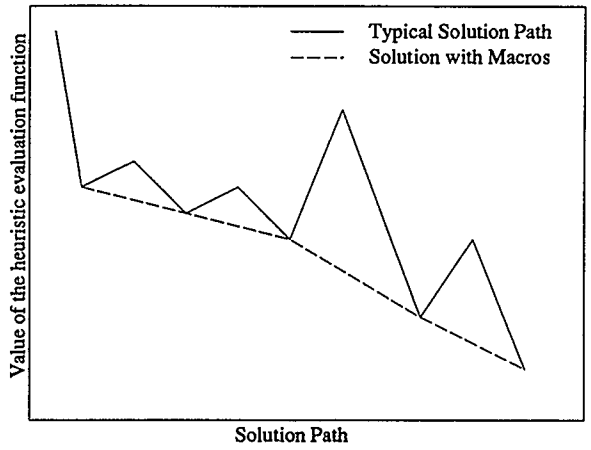


FIGURE 3.7. The Effect Of Macros Learned By The Minimum To Minimum Heuristic

3.2.3. Static Filter. The Static Filter uses three tests. It removes new macros that are duplicates of existing operators. The filter also removes macros that are longer than a given threshold. Long macros will usually have complex preconditions and thus will not often be applicable. Chapter 4 examines this motivation in more detail. Lastly, a domain dependent test may be applied.

A macro that passes these tests is immediately added to the operator list, allowing *in-trial* learning as well as cumulative learning across trials. Macros learned early in the search can be beneficial for the remaining solution as well as for later problems.

3.2.4. Dynamic Filter. MACLEARN also uses a Dynamic Filter to remove macros that are seldom or never used. This is invoked to remove unused macros after a training session. It proves to be effective in the Peg Solitaire domain, but it may not always be helpful, since it can remove potentially useful macros.

3.2.5. The Minimum To Minimum Heuristic. MACLEARN uses a *Minimum to Minimum* heuristic to find new macros. (Figure 3.7). When the value of a heuristic function at a given state is less than the value at its children, a new macro is proposed

between the previous and current minima along the solution path.[1] The motivation behind this heuristic is to eliminate the segments of the search where the heuristic function is increasing. When a solution must follow such a segment, the neighboring region of lower heuristic value is exhaustively searched. By remembering macros that traversed the "hills," the system tries to avoid these more expensive segments. The macros make the heuristic function monotonically decrease along the solution path.

3.3. Results

To test his system, Iba ran three experiments in the Peg Solitaire domain. The first examined the value of learning macros. The second tested the effects of cumulative learning over several problems. The third tested the static filtering heuristics described in Section 3.2.3.

For the first test, each of the problems in Figure 3.8 were attempted without macros, then again with macro learning. The trial with macro learning was repeated to allow the use of macros that were learned near the end of problem solving. Then the Dynamic Filter was used to remove unused macros, and the trial was performed again. This experiment showed that macro learning improved the speed of problem solving for the smaller puzzles and allowed some of the larger puzzles to be solved where they could not be solved within resource limits by heuristic search alone.

Even with in-trial macro learning, not all of the Peg Solitaire problems were solved. The second experiment tested the effects of cumulative learning across the problems in Figure 3.8. At the end of the first pass through all of these problems, the Dynamic Filter was invoked to remove unused macros. Then each problem was attempted again, and the Dynamic Filter was invoked after *each* one. This allows macros to

[1] Iba [13] describes the heuristic function as *increasing* towards the goal rather than decreasing, and thus the macro learning heuristic is seen as "Peak to Peak."

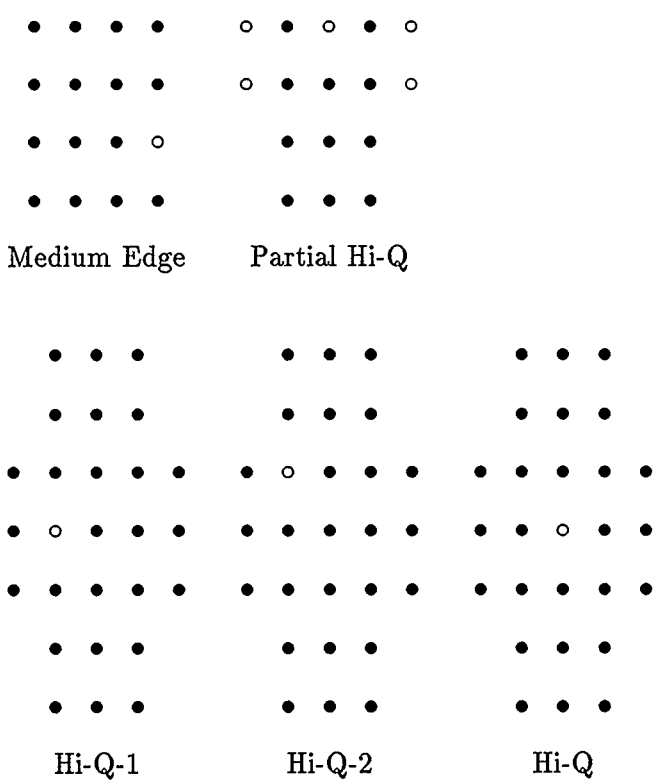


FIGURE 3.8. Peg Solitaire Problems Used For Iba's Experiments

be generated in one problem and used in solving another one, but keeps control of the number of macros. This experiment showed that cumulative learning allowed MACLEARN to solve problems that it could not solve with in-trial learning alone.

The final experiment tested the static filtering mechanism (3.2.3). Again, the problems in Figure 3.8 were used as a training sequence. To test each component of the Static Filter, this training sequence was attempted with all of the components disabled, then with each individual component enabled, then with all components enabled. The results showed that the static filtering mechanism is useful, and that each of the components resulted in an improvement in problem solving speed.

3.3.1. Discussion. Iba makes several implicit assumptions about the problem domain that have an effect on the search. MACLEARN assumes that the heuristic evaluation function is non-monotonic. If the evaluation function is monotonic, no macros will be generated. In practice this is not really important, since best first search already works well in a domain with a monotonic evaluation function. The representation of macros as rectangular arrays may have an important effect on the search, which is not discussed by Iba. MACLEARN works well for puzzles that can easily be defined on rectangular grids, but there are many important problems that do not involve rectangular arrays.

Despite its limitations, the MACLEARN system can result in significantly better performance than heuristic search alone. The ability to learn macro operators on simple training problems allows MACLEARN to solve several difficult problems.

CHAPTER 4

Optimal Tunneling

In a domain such as Peg Solitaire, the Minimum to Minimum heuristic finds many useful macros because the distances between minima are small. However, in other domains the minima are much further apart, and the Minimum to Minimum heuristic proposes much longer, less useful macros. This chapter describes a new heuristic for learning macros: *Optimal Tunneling*. The intuition behind Optimal Tunneling is described in Section 4.1. This intuition is examined in Sections 4.2 to 4.4. Chapter 5 shows the results of several tests comparing the Minimum to Minimum heuristic with Optimal Tunneling.

4.1. Sokoban

SOKOBAN [11] is an interesting family of puzzles in which minima are far apart. It is loosely related to the Eight puzzle (Figure 2.1), but is more complex, with a domain consisting of several objects (Figures 4.1 and 4.2): a penguin, some balls and the same number of goal squares, and some walls that form a maze.

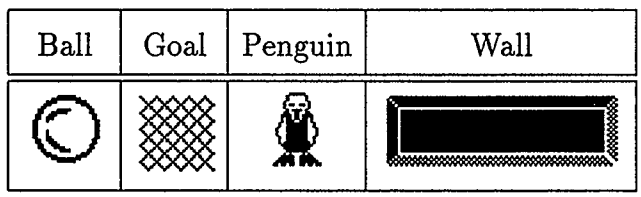


FIGURE 4.1. The Parts Of SOKOBAN

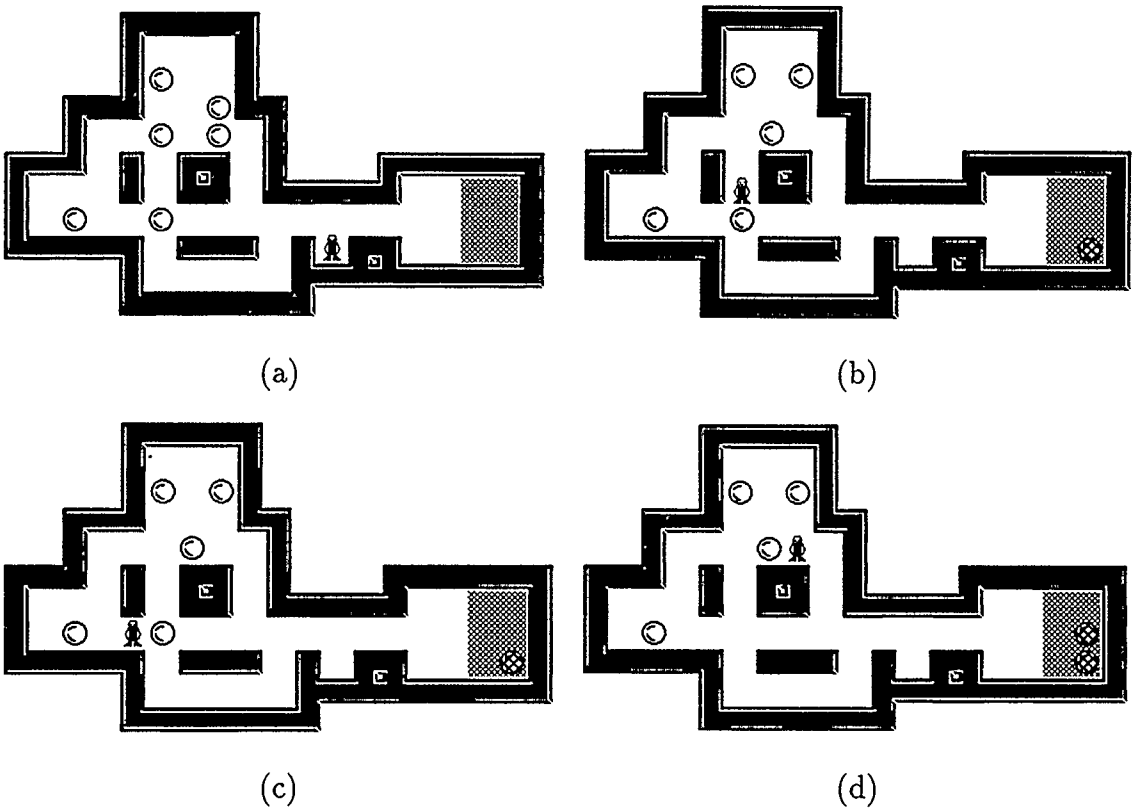


FIGURE 4.2. The Solution To A SOKOBAN Problem

The object of the game is to move the penguin — up, down, left, and right — to push all of the balls onto the goal squares. The penguin can be moved by itself or push exactly one ball into an empty space or goal.

4.1.1. Heuristic Functions For Sokoban. A heuristic function for SOKOBAN might be based on the one Iba used for the tile sliding domain [13]. The function would return a vector with the following components:

(1) The number of balls not in goal locations.

(2) The average Manhattan[1] distance of each ball from each of the empty goal squares.

(3) The Manhattan distance of the penguin from the closest ball not in the goal.

The elements of this vector would be compared in this order and suggest a set of subtasks to be solved.

Figure 4.2 shows an initial state and three stages of solution. In stage (b), the problem solver has reached an impasse. It has only two moves available to it, and both result in an increase in the heuristic function. This stage is a minimum. To move the ball closer to the goal, the penguin must first be moved away from the ball, around the wall, and back towards the ball to stage (c). From stage (c), the heuristic function gradually decreases as the ball is pushed into the goal. The penguin is then moved until it is adjacent to the nearest ball not in the goal, stage (d). Stage (d) is also a minimum, since the only available moves are to push the ball away from the goal, or to move up or right, both of which are away from the ball. At stage (d), the Minimum to Minimum heuristic would propose a macro from stage (b) to stage (d). This macro is composed of 32 primitive steps, and is far too specific to be useful.

A much more useful macro would be the one from (b) to (c) since it could be used many times during the solution to this screen. Notice that the heuristic function returns the same vector for each of these states.

So our intuition suggests macros from a minimum in the heuristic function to the next state where the heuristic value is the same as this minimum (but not necessarily at a new minimum). We now examine this intuition.

---

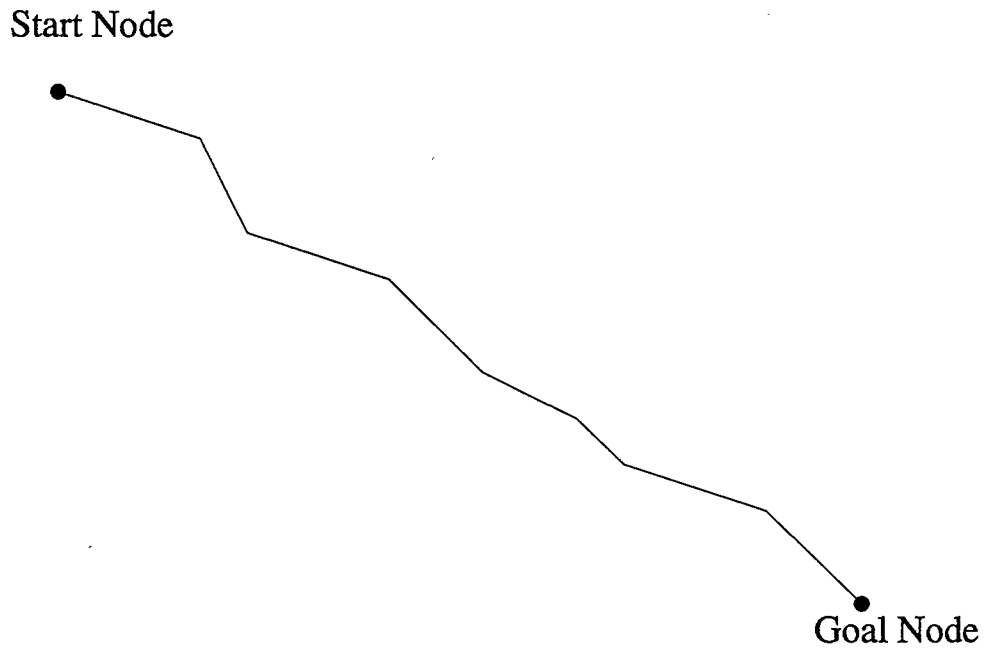
[1]The *Manhattan* distance between two points is the number of grid steps needed to go from one to the other. Formally, $MD(x_1, y_1, x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$.

4.2. Search As "Pouring Water"

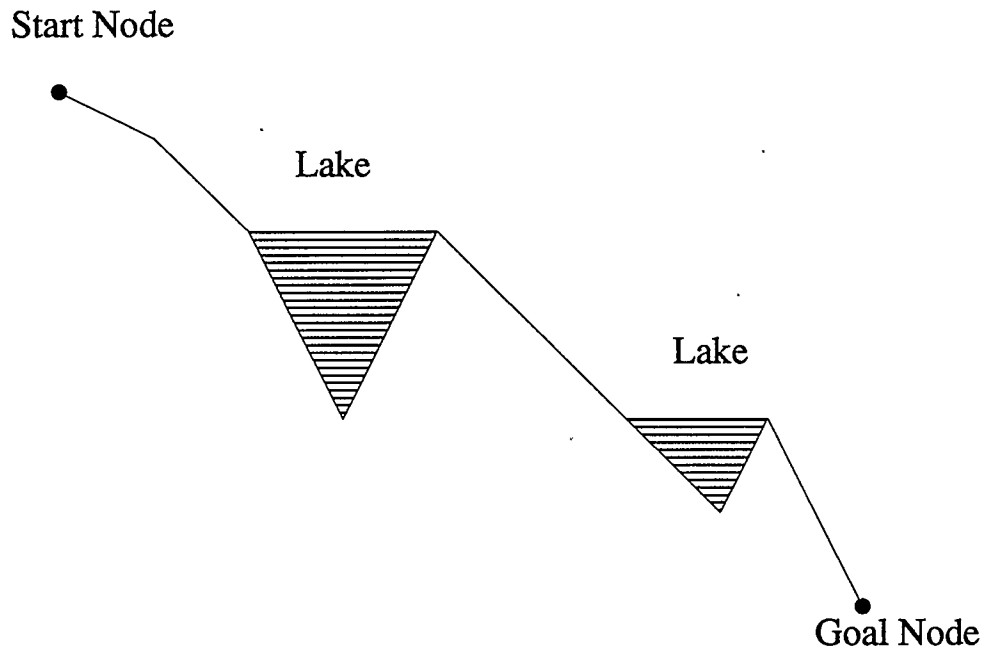
Consider a two dimensional search space. The heuristic function defines a three dimensional surface over this space. If one were to "pour water" at the start state on the surface, it would run downhill until it reached the goal, and the path of the wave front represents the progress of the best–first search (water takes the path of least resistance). Figure 4.3(a), shows an example of the heuristic function value along a solution path. However, if there is a valley along this path, the water must fill the valley to form a lake before continuing on down the next slope, as in Figure 4.3(b). In a best first search, this "lake" represents the work needed for the problem solver to search each state in the valley until it finds a way out. The search must look at all the states that the heuristic function estimates are better, ie. the neighbors in the "valley," before the search can continue over the valley ridge.

4.3. Optimal Tunneling Macros Reduce Search Cost

Adding a macro to the operator set allows the heuristic search to avoid examining all the states in the valley. If a macro can be applied from a state in a valley it can allow the system to jump to a state outside the valley in a single step. A macro acts like a tunnel or pipe that allows the water to drain. Figure 4.4(a) shows the same search space as Figure 4.3(b) with macros added using the Minimum to Minimum heuristic. These "tunnels" allow the water to drain preventing any "lakes" from forming. However, they are longer than they need to be. Figure 4.4(b) does the same job with the Optimal Tunneling heuristic using shorter tunnels. Having simpler preconditions, the shorter macro is more likely to be used in similar situations. Iba's static filtering mechanism (Section 3.2.3) deals with the problem of long macros by preventing them from ever entering the operator pool. The Optimal Tunneling method tries to avoid

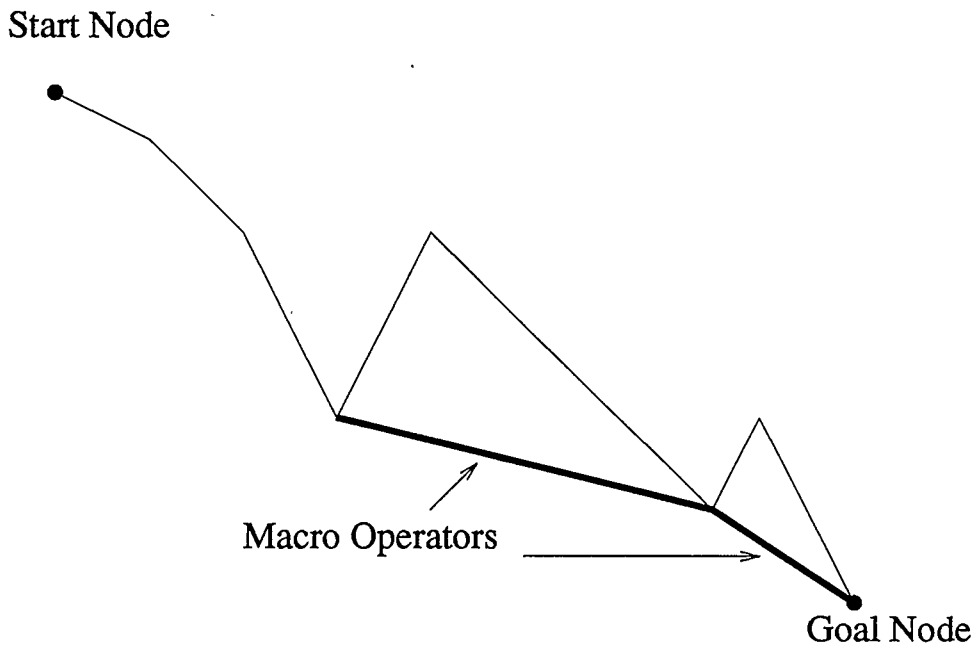


(a) Monotonic Search

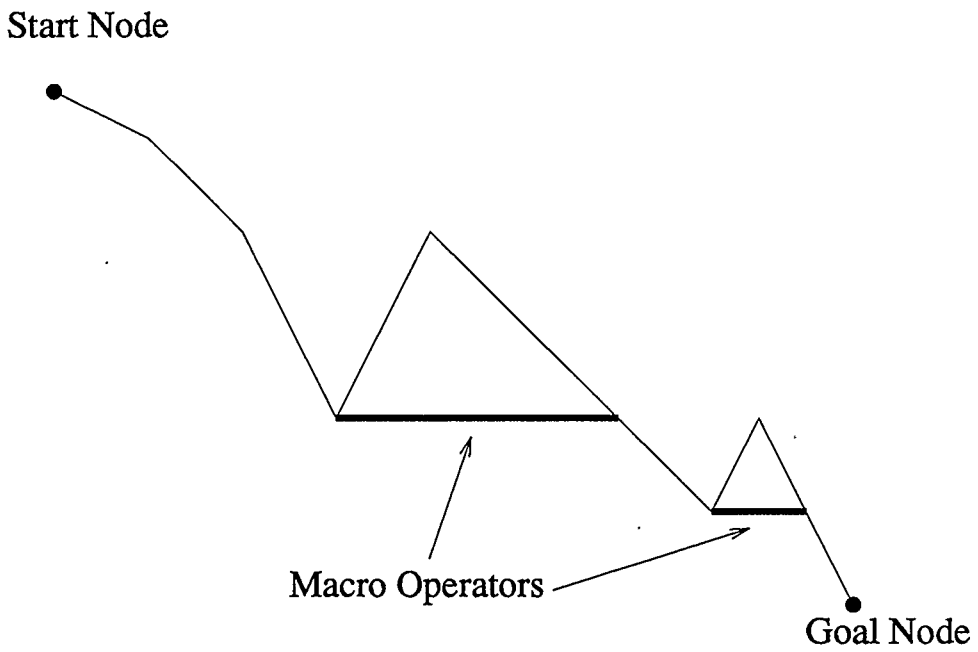


(b) Search with Valleys

FIGURE 4.3. The Effects Of "Pouring Water" On Example Search Spaces



(a) Minimum to Minimum



(b) Optimal Tunneling

FIGURE 4.4. Macros As Tunnels

proposing long macros in the first place. Although the shorter macros mean more steps in the solution, the added steps cost little in the search, since they occur in a segment where the heuristic function decreases. The Optimal Tunneling heuristic proposes macros from a valley floor (a minimum in the heuristic function) to the next state outside the valley at the same height.

From the water pouring analogy it is easy to see why macros should be proposed starting from a minimum. If the heuristic search without macros ever enters a valley, it will eventually reach the valley floor unless it finds the goal somewhere on its way down. If a macro were proposed starting from part way up the valley side, there is no guarantee that the state on the valley side will be found immediately. More likely, the search will reach the valley floor and then have to search every state in the valley until it finds the starting point for the macro. This suggests that the "Optimal Aqueduct" heuristic (Figure 4.5) will not work well. The search may get lucky and hit the start of the aqueduct, but it is more likely to search the entire valley. Referring back to the water pouring analogy and real tunnels for water, most well designed city storm sewers have their entrances at local minima. Any minimum that does not have a storm sewer entrance will flood when it rains.

The water pouring analogy also suggests that a horizontal tunnel is the shortest tunnel that will allow water to drain from the valley. Any shorter macro would give an uphill slope to the tunnel as in Figure 4.6.

The problem with this macro is that the heuristic search has no reason to select it over any other operator. When expanding the state at the floor of the valley, the primitive operators will result in states with slightly higher heuristic values than the valley floor. The macro, on the other hand, gives a state with a much higher heuristic value. As a result, the best first search will choose to expand the states inside the

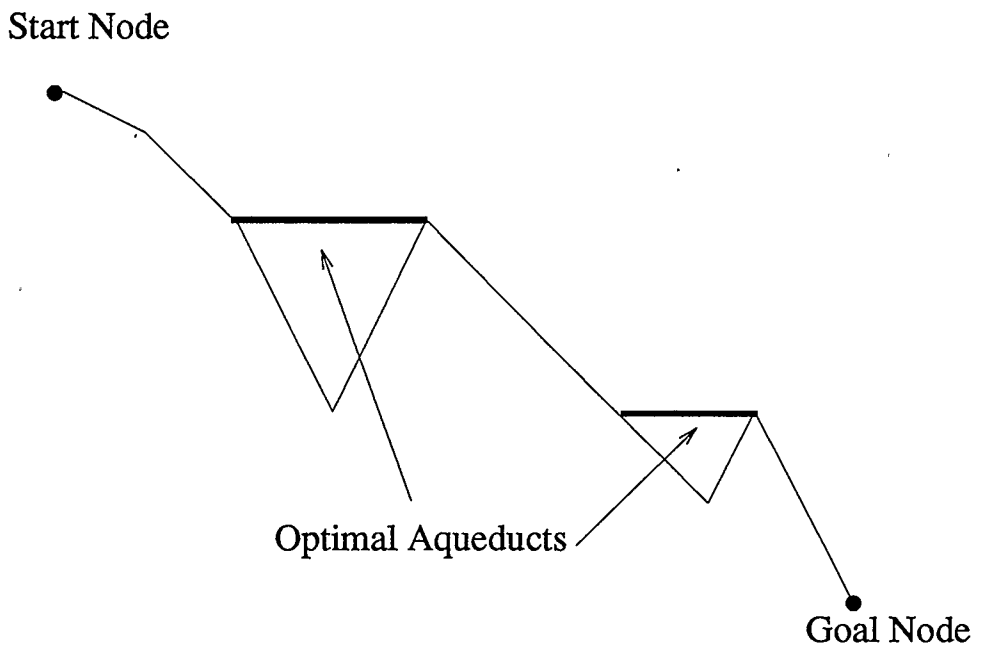


FIGURE 4.5. The Optimal Aqueduct Heuristic

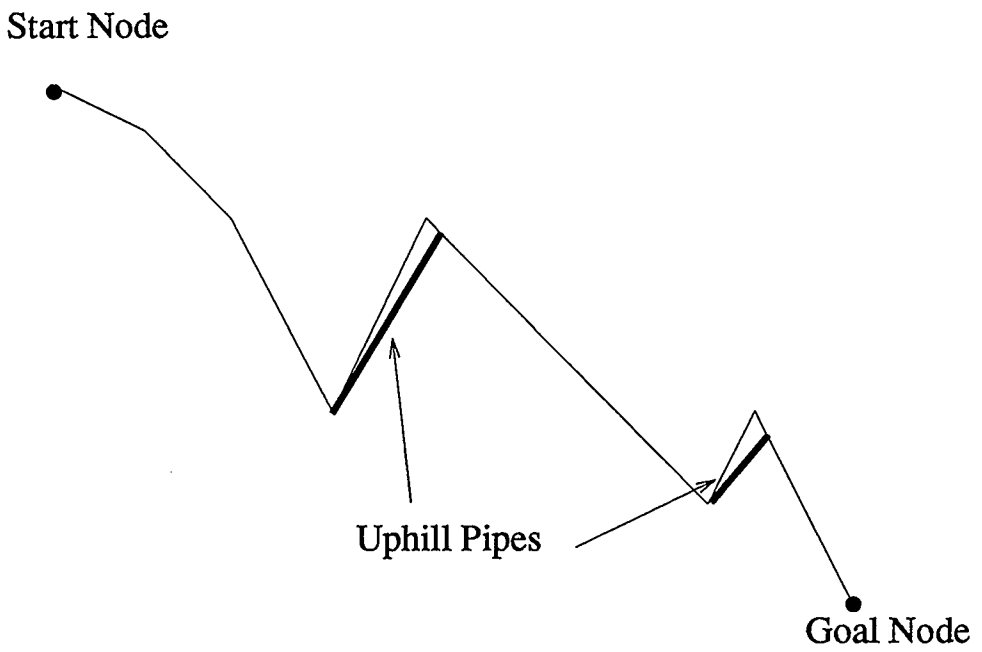


FIGURE 4.6. Uphill Pipes

valley first. The state generated by the macro would only be expanded after all lower states in the valley had been exhaustively searched. The water would fill up the valley until the level in the valley had reached the height of the top of the pipe.

The shortest macro that forces the search to continue outside the valley is the one generated by the Optimal Tunneling heuristic. When the state at the valley floor is expanded, the state generated by the macro will have a lower heuristic value than the states generated by the primitive operators in the valley. The heuristic search will select this new state first since it has the lowest heuristic value.

4.4. Optimal Tunneling Macros Improve Heuristic Function Accuracy

DEFINITION 4.1. *$f(n)$ is the minimum cost from state n to a goal.*

DEFINITION 4.2. *$f^*(n)$ — the heuristic function — is an estimate of $f(n)$.*

It is expected that the more nearly f^* approximates f, the better the algorithm will do [3]. Unfortunately, for most problems, f^* is not identical to f. How can the difference be reduced?

Macros do not change the function f^*, but do change f. New macros change the search space by reducing the cost of moving from one state to another. Since we are looking for any solution to a puzzle, not necessarily the shortest one, we can assume that the cost of a macro operator is 1, the same as the cost of a primitive operator. If it is possible to move from state a to state b using a macro operator M, then $f(a) \leq f(b) + 1$ since if the minimum cost of moving from b to the goal is $f(b)$, the minimum cost of moving from a to the goal cannot be greater than the cost of moving to b, and then moving to the goal. So if $f^*(a) < f(a)$, learning a macro from a could reduce the difference between the two.

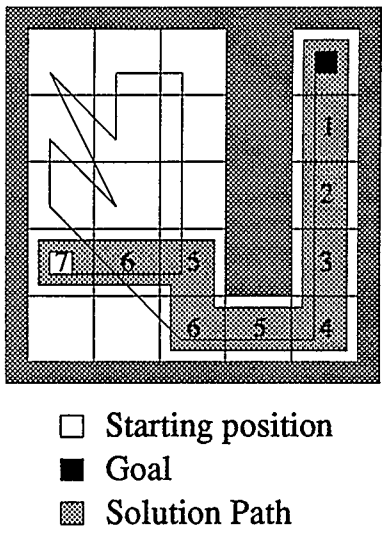


FIGURE 4.7. The Solution Path Without Macros

The most useful macro to learn from a is one that makes $f(a) = f^*(a)$. Such a macro should move from a to some b such that $f(b) = f^*(a) - 1$. In other words, the macro should move us from state a to a state in which the minimum distance to the goal is $f^*(a) - 1$. In practice, finding the state b may be computationally difficult. However, we can find b' such that $f^*(b') = f^*(a) - 1$. Here we find a state b' where the value of the heuristic function is one less than the value of the heuristic function at state a. The Optimal Tunneling heuristic finds macros where $f^*(b') = f^*(a)$. The macros that it learns change the distance from state a to the goal so that it is one more than the value of the heuristic function at state a.

An example of a macro that changes the search space is given in Figures 4.7 to 4.10. Figure 4.7 shows a solution to a simple path-finding problem with no macro operators. The Manhattan distance from each state along the solution path to the goal is shown in the figure. The number of steps needed to reach the goal from the start state is nine, but the heuristic estimates that it is seven. Figure 4.8 shows the estimated and actual distances to the goal at each step along the solution path.

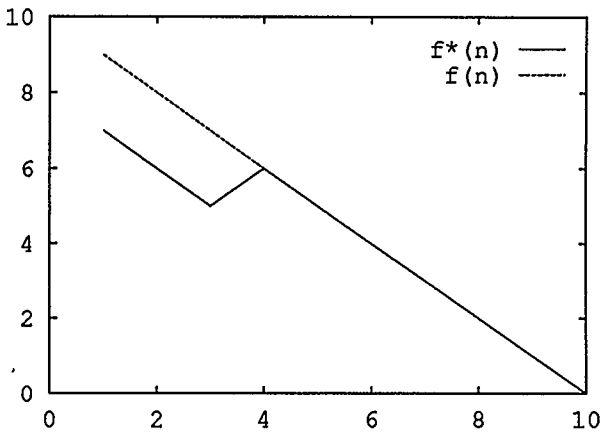


FIGURE 4.8. The Estimated, $f^*(n)$, And Actual, $f(n)$, Distance To The Goal Without Macros

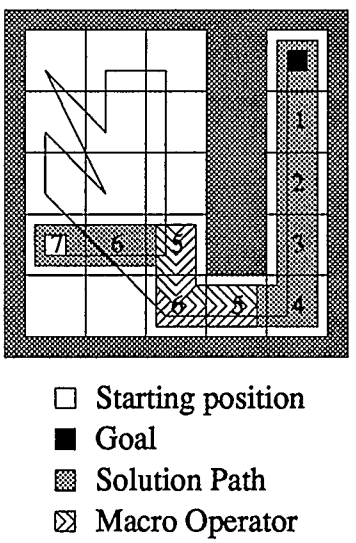


FIGURE 4.9. The Solution Path With Macros

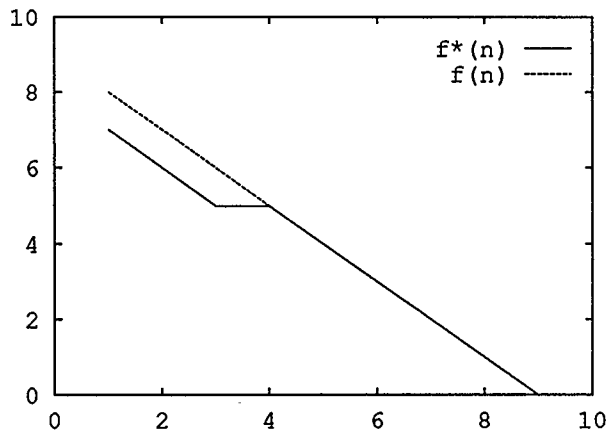


FIGURE 4.10. The Estimated, $f^*(n)$, And Actual, $f(n)$, Distance To The Goal With Macros

Learning a macro changes the search space by allowing a move across the two states with a heuristic value of five in one step (Figure 4.9). The heuristic function has not changed, but the addition of the macro operator changes the number of steps needed to reach the goal to eight. This macro has changed the search space to make the heuristic function more accurate.

4.5. Summary

The Minimum to Minimum heuristic works well for domains in which minima are close together. SOKOBAN is an example of a domain where minima are far apart. Minimum to Minimum does not learn useful macros in this domain. An example suggests learning a macro from a minimum to the next state with the same heuristic value. The Optimal Tunneling heuristic learns macros in this way.

A water pouring analogy suggests that "valleys" along the solution path will slow the progress of a best first search. Macro operators act like "tunnels," allowing water to drain from a valley without filling it to form a "lake." The Minimum to Minimum heuristic proposes macros that allow the water to drain from one valley floor to another valley floor, but the macros are longer than they need to be. The water pouring analogy suggests that a macro should start at a minimum, and end at a state with the same heuristic value. The Optimal Tunneling heuristic proposes these macros, which allow the water to drain, but with as short a tunnel as possible.

The performance of a best first search depends on the accuracy of the heuristic function. Macro operators change the search space, and can potentially make the heuristic function more accurate. The Optimal Tunneling heuristic finds macros that do this.

Since the Optimal Tunneling heuristic finds macros that tend to make the heuristic function more accurate, these macros should improve the efficiency of the search.

Optimal Tunneling also finds the shortest macro that avoids searching "valleys." This suggests that in general, Optimal Tunneling will perform better than the Minimum to Minimum Heuristic. The results in Chapter 5 confirm the superiority of Optimal Tunneling.

CHAPTER 5

Results

5.1. Implementation Status

I have implemented a problem solver based on Iba's work. It is written is Chez Scheme, and supports both the Minimum to Minimum and the Optimal Tunneling heuristic. Source code for the system is available from `jamesm@cpsc.ucalgary.ca`, or `bruce@cpsc.ucalgary.ca`. During the course of my research, I also implemented a problem solver based on Korf's MPS [15] (Section 2.3.3).

The structure of the problem solver is similar to the model described by Iba (Section 3.1). The core of the system is a heuristic search. Each operator is stored as a list of "before" and "after" pairs for each possible orientation of the operator. The "before" and "after" patterns are stored as a list containing a list of symbols for each line of the pattern. The operator pool is stored as a simple list of operators.

Several data structures are used to improve the performance of the system. Each newly generated state must be checked to see if it was generated previously. This is done by storing each state in a hash table [10], which can be quickly checked to see if a new state is a duplicate. Another potentially slow operation is the maintenance of the sorted list of open states. These states are kept in a heap [10], providing $O(\log n)$ time for an insertion, and $O(\log n)$ time for the removal of the minimal element.

The pattern-matching algorithm is not as efficient as it could be. Since it tries

to match a pattern at each possible position without using any of the information from previous positions, it requires $O(m \times n)$ comparisons for an operator with m symbols, and a state description with n symbols. Amir *et al* [1] present a more efficient algorithm. The Rabin-Karp string matching algorithm [5] can also be generalized to 2-dimensional pattern matching giving $O(n + m)$ comparisons.

5.2. Problems With Heuristic Search

When the heuristic function is one to one, the heuristic search is completely defined. However, if two states can have the same heuristic value, then the heuristic search must choose which to expand first. Such choices early in the search can have drastic effects on the outcome. The choice of which state to select among those with the same heuristic value is usually defined implicitly by the problem solving system, and is affected by several design choices:

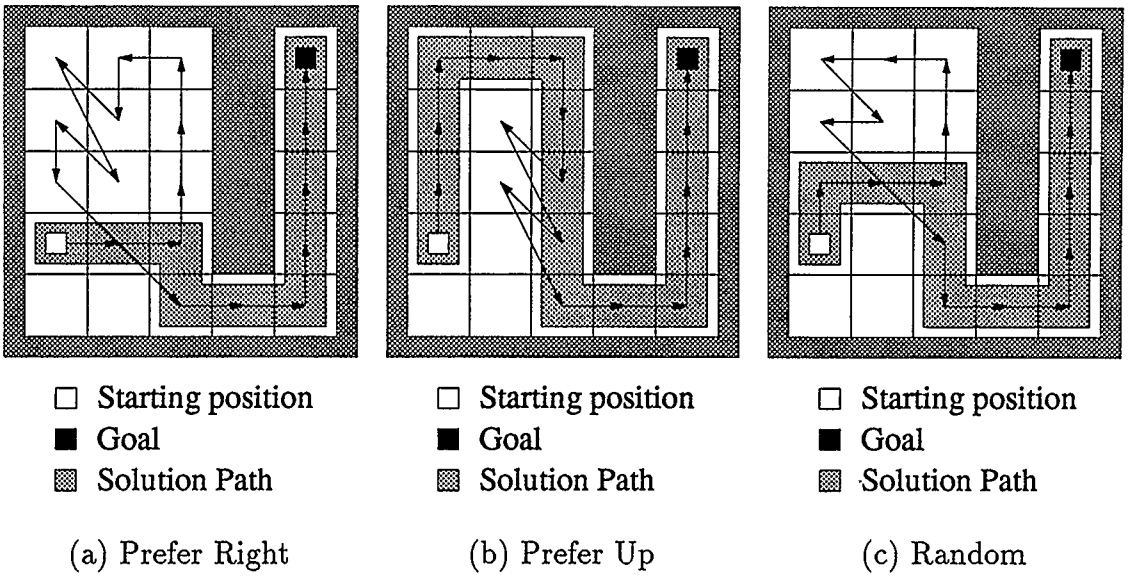
(1) What order are operators applied to a state being expanded?

(2) If operators can be matched when rotated or reflected, in what order are the rotations or reflections applied?

(3) When new states are added to the list of unexplored states, are they added before or after states with the same heuristic value?

(4) Does the data structure used to store the unexplored states maintain the order of states with the same heuristic value?

Figure 5.1 shows the effects of various choices on a simple path finding domain. Part (a) shows the effects of examining the state generated by the "Right" operator before considering others. Part (b) shows the effects of preferring the "Up" operator. For part (c), when two states have the same heuristic value, one is chosen at random. Although all three examine a similar number of states, the solutions found are signif-



(a) Prefer Right (b) Prefer Up (c) Random

FIGURE 5.1. The Effects Of Operator Choice On Heuristic Search

icantly different. The order in which states are expanded can change both the total number of states expanded during a search, and the solution path.

These effects are compounded by macro learning. Since these choices will cause the search to take different paths through the search space, different macros will be learned, which increases the difference between the two searches. Also, the addition of macros adds new choices when heuristic values are the same:

(1) Are macro operators expanded before or after primitive operators?

(2) Are new macros expanded before or after old ones?

These choices can be made arbitrarily in that there is no clear reason to prefer any one of them, but their effects can significantly alter the results of a given trial. Since Iba does not specify what choices he makes for his system, it is difficult to reproduce his results exactly. My system makes choices as described in Appendix A.

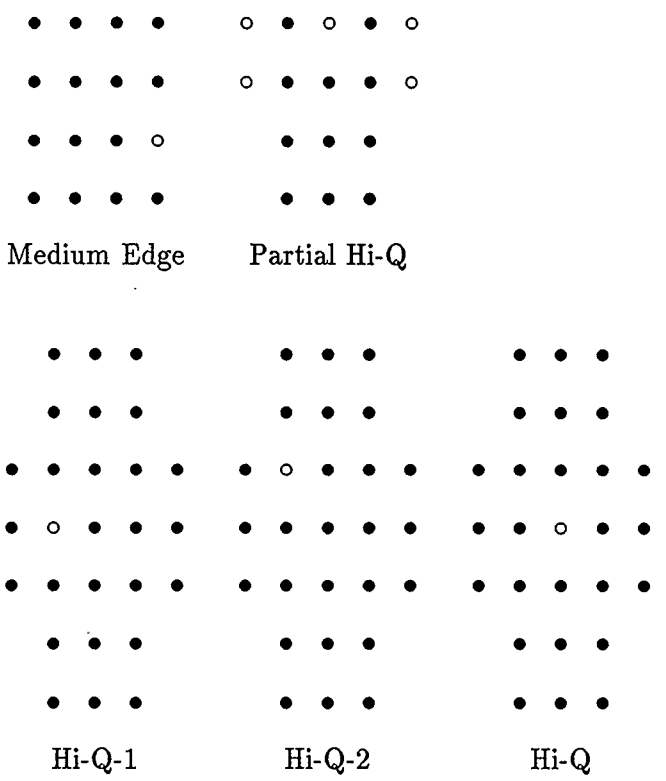


FIGURE 5.2. Problems Used For The Peg Solitaire Experiments

5.3. Comparative Tests

The Optimal Tunneling heuristic and the Minimum to Minimum heuristic are compared using the problem solver described above. For each comparative test, the only change made is the choice of macro proposing heuristic. The tests were performed on a SparcStation 10 Model 30 with 32Mb of physical memory, and 252Mb of virtual memory.

5.3.1. Peg Solitaire. Iba tested the MACLEARN system on the Peg Solitaire domain. Since the minima are close together, this is a domain for which the Minimum to Minimum heuristic performs well. Even in a domain where Minimum to Minimum performs well, the Optimal Tunneling heuristic shows a significant improvement.

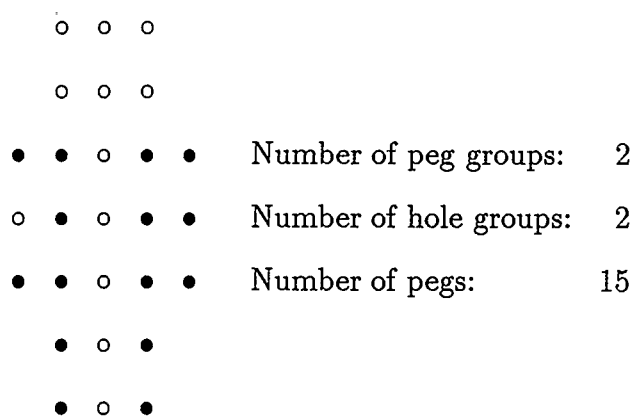


FIGURE 5.3. A Peg Solitaire State Whose Evaluation Function Value Is (2,2,15)

5.3.1.1. *Heuristic Function.* The Heuristic Function used for Peg Solitaire was identical to the function used by Iba. It is a vector with three components:

(1) The number of groups of pegs

(2) The number of groups of holes

(3) The number of pegs

Groups are defined as horizontally or vertically adjacent sets. Figure 5.3 shows an example state, and its evaluation.

5.3.1.2. *Static Filter.* For the tests in the Peg Solitaire domain, the static filter included all of the elements in Section 3.2.3. The threshold for the length test was seven.

The domain-specific test for Peg Solitaire was a *connectedness* test. This test rejects macros in which the pegs are not connected in the "after" side.

5.3.1.3. *Experiment 1.* The first experiment compares Minimum to Minimum with Optimal Tunneling for three different parts of a learning trial. Each of the problems in Figure 5.2 was attempted three times. As in Iba's Experiment 1 (Section 3.3),

the problems were attempted twice with macro learning and then again after dynamic filtering. Each problem was attempted separately from the others; there was no cumulative learning across different problems. The only difference between this experiment and Iba's experiment is that we do not attempt to solve the problems without macro learning since we want to compare the two macro-learning heuristics.

Figure 5.4 shows the results of the first trial with macro learning. Neither system solved the Hi-Q puzzle within the 10000 second time limit without cumulative learning. For the Hi-Q-1 problem, Optimal Tunneling was only slightly faster than Minimum to Minimum, but Optimal Tunneling was significantly better for all the other cases.

For the second try with macro learning (Figure 5.5), the results were more varied. For the Partial Hi-Q and the Hi-Q-2 problems, the solutions for Optimal Tunneling were found without backtracking, and much more quickly than those for Minimum to Minimum. The Medium Edge problem was actually solved monotonically (without backtracking) by Minimum to Minimum, but since it had a higher branching factor, it took slightly longer than Optimal Tunneling which needed to backtrack. In the final phase of the experiment, the dynamic filter reduced the branching factor enough that Minimum to Minimum could solve it faster. The only case where Optimal Tunneling was worse than Minimum to Minimum was Hi-Q-1. Neither heuristic solved this problem as quickly as the other problems, and Optimal Tunneling took nearly twice as long as Minimum to Minimum.

After Dynamic Filtering (Figure 5.6), Medium Edge was the only problem that was solved more quickly by Minimum to Minimum. This was the simplest of the problems, and both systems solved it in less than one second. Optimal Tunneling performs more effectively on the more complex problems. Partial Hi-Q and Hi-Q-2

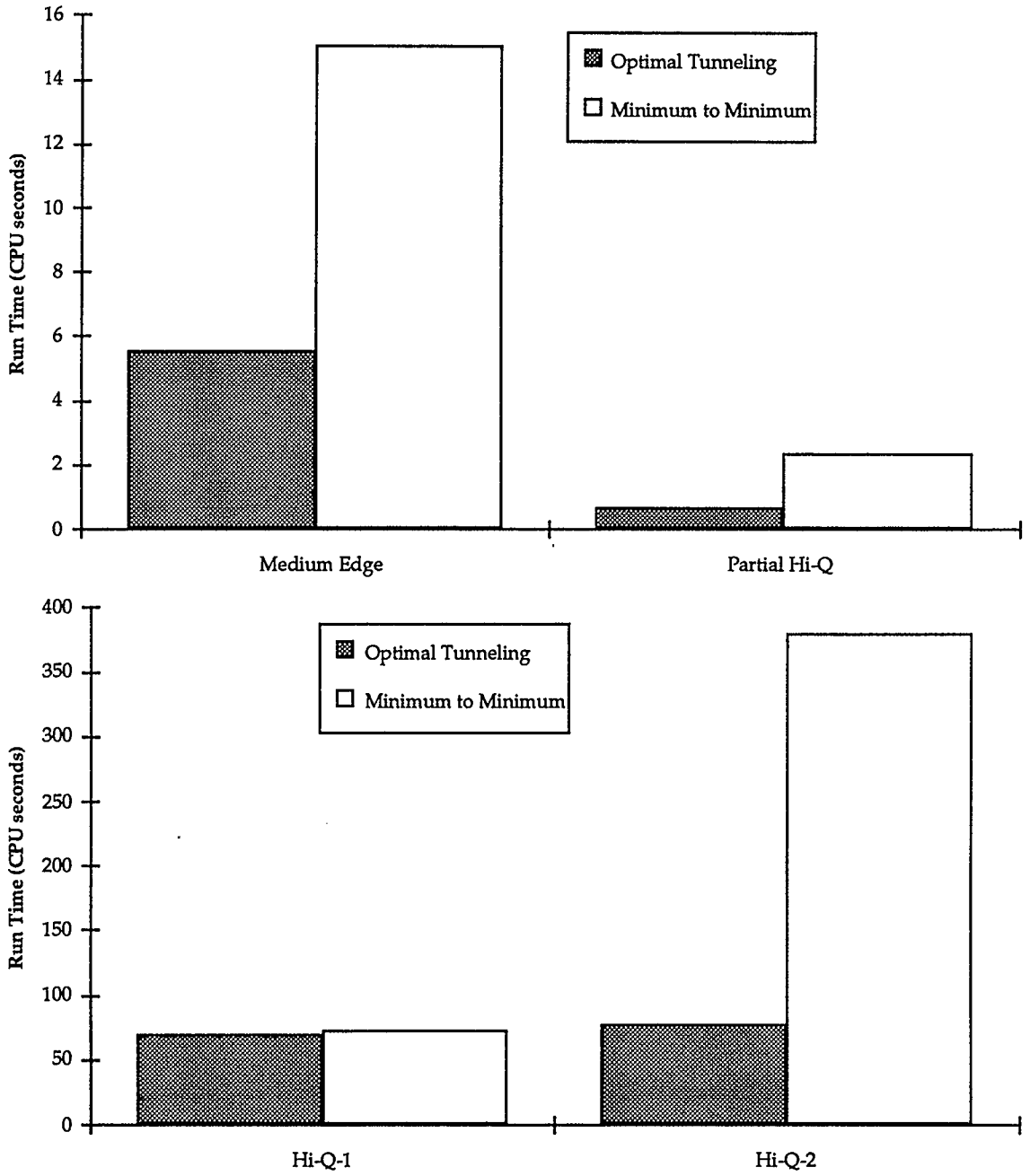


FIGURE 5.4. First Try With Macro Learning For Peg Solitaire

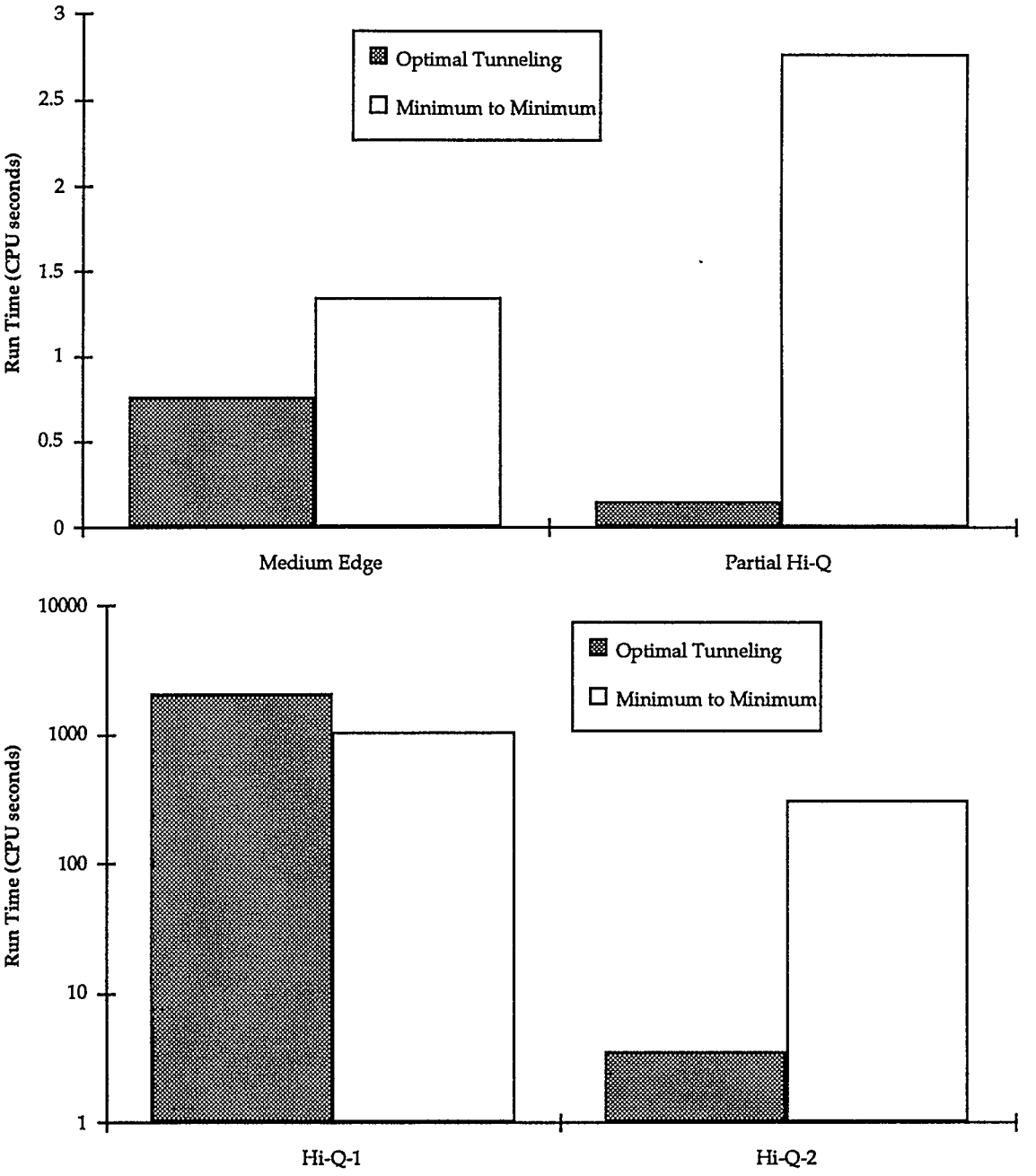


FIGURE 5.5. Retry With Macro Learning For Peg Solitaire

were both solved monotonically and took about 1/5th as much time as Minimum to Minimum. Hi-Q-1 seemed to give both systems some trouble, but again, Optimal Tunneling was nearly 5 times faster than Minimum to Minimum.

Although a few problems took more time with Optimal Tunneling, it generally performs much better than Minimum to Minimum for problems in the Peg Solitaire domain without cumulative learning.

5.3.1.4. *Experiment 2.* The second experiment with Peg Solitaire compares the two macro-learning heuristics for cumulative learning across similar problems. For this test, all of the problems in Figure 5.2 except the full Hi-Q were used as a training sequence. Hi-Q was not solved by Minimum to Minimum even with cumulative learning, so it was left out of this test[1]. At the end of the first pass through all of these problems, the Dynamic Filter was invoked to remove unused macros. Then each problem was attempted again, and the dynamic filter was invoked after *each* one. Figure 5.7 shows the run time for both macro learning techniques. Another useful measure of performance is the *cumulative run time*, since it takes into account the work needed to learn macros during the initial training sequence. Figure 5.8 shows that Optimal Tunneling results a clear improvement in cumulative run time over Minimum to Minimum.

For the Peg Solitaire domain, both with and without cumulative learning, Optimal Tunneling performs significantly better than Minimum to Minimum.

5.3.2. Tile Sliding. Another domain used to compare the Optimal Tunneling heuristic to the Minimum to Minimum heuristic is Tile Sliding. Tile Sliding is a name given to the family of puzzles that includes the Eight puzzle and the Fifteen puzzle.

[1]With cumulative learning Hi-Q was solved by Optimal Tunneling, but it is not worthwhile to use it in the training sequence since Minimum to Minimum failed to solve it.

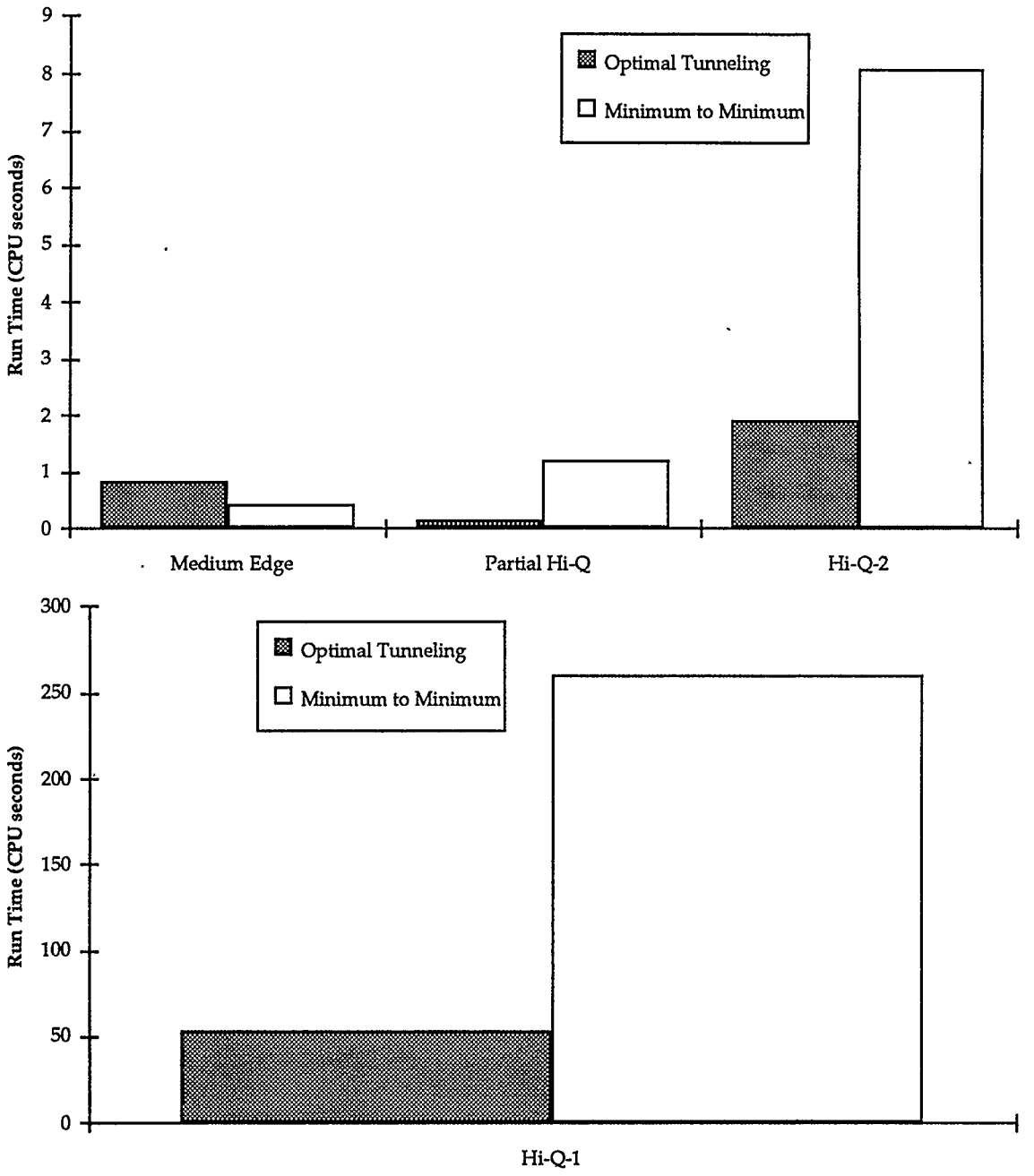


FIGURE 5.6. After Dynamic Filtering For Peg Solitaire

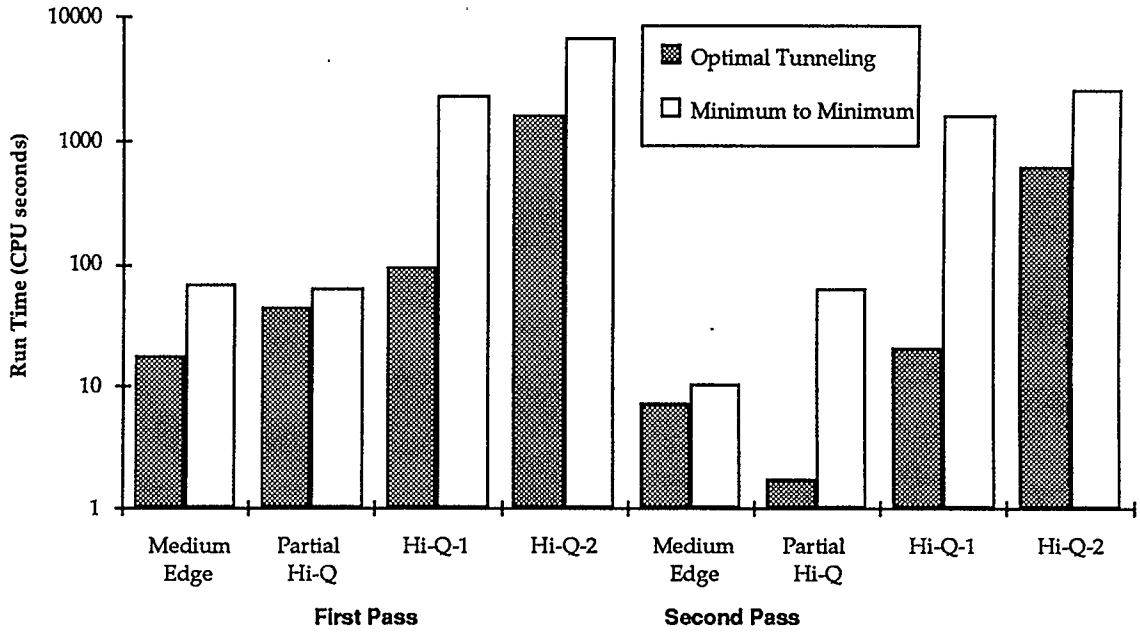


FIGURE 5.7. Run Time For Peg Solitaire Experiment 2

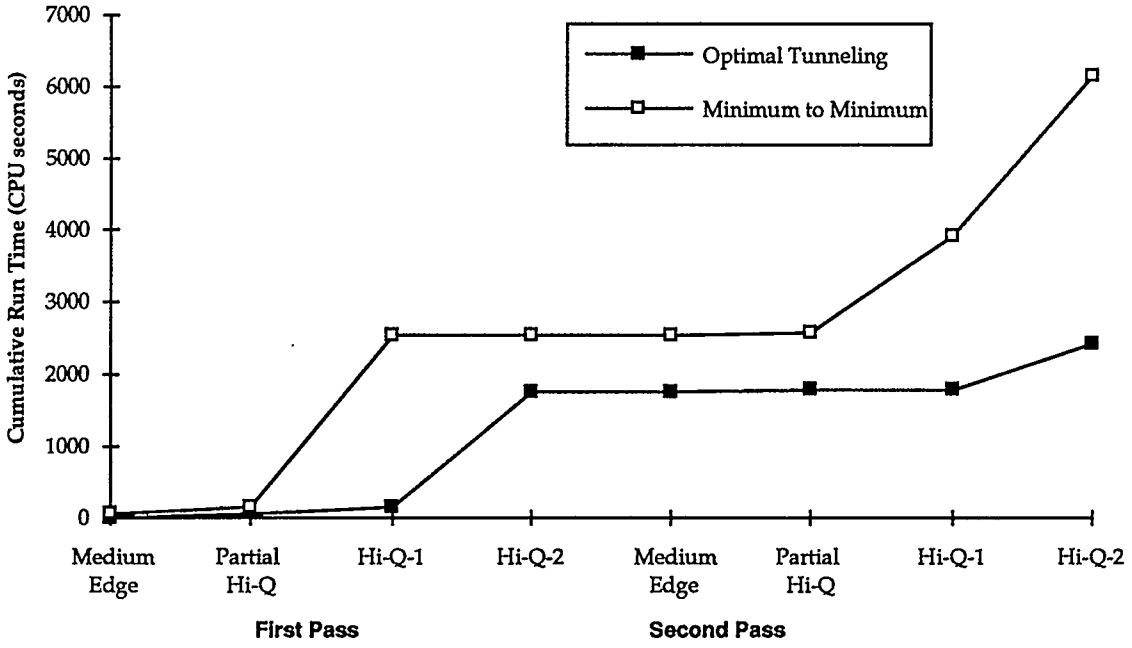


FIGURE 5.8. Cumulative Run Time For Peg Solitaire Experiment 2

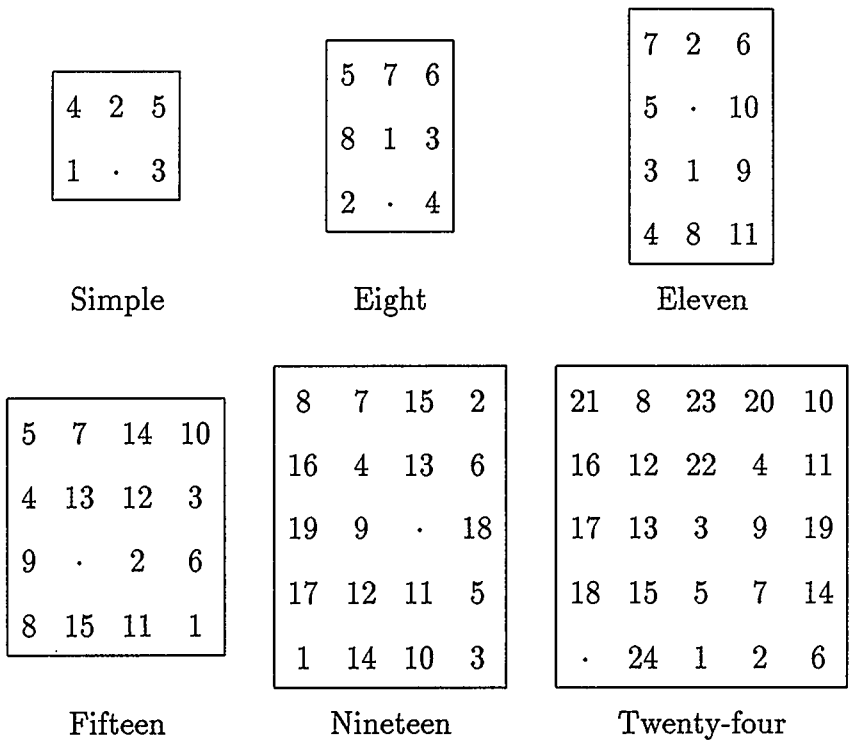


FIGURE 5.9. The Problems Used For The Tile Sliding Experiment

Figure 5.9 shows the problems used for the experiment in the Tile Sliding domain. Each puzzle was chosen at random, but was fixed so that identical problems were attempted using each heuristic. The tests were similar to the second experiment for Peg Solitaire. All of the problems were used as a training sequence. At the end of the first pass through all of these problems, the dynamic filter was invoked to remove unused macros. Then each problem was attempted again, and the dynamic filter was invoked after *each* one.

5.3.2.1. *Heuristic Function.* The heuristic function for the tests in the tile sliding domain was identical to the one used by Iba. The function returns a vector with the following components:

(1) The number of consecutive tiles in their goal locations multiplied by negative one. Counting starts from the first tile and proceeds until the first mismatch is encountered. This number is then multiplied by negative one so that the value gets smaller as it approaches the goal.

(2) The Manhattan[2] distance of next tile to be placed from its goal square.

(3) The Manhattan distance of the blank from the next tile to be placed.

The elements of this vector would be compared in this order and suggest a set of subtasks to be solved.

5.3.2.2. *Static Filter.* For the tests in the Tile Sliding domain, the static filter included only the redundancy and length tests from Section 3.2.3. No domain-dependent test was used. The threshold for the length test was thirty.

[2]The *Manhattan* distance between two points is the number of grid steps needed to go from one to the other. Formally, $MD(x_1, y_1, x_2, y_2) = |x_1 - x_2| + |y_1 - y_2|$.

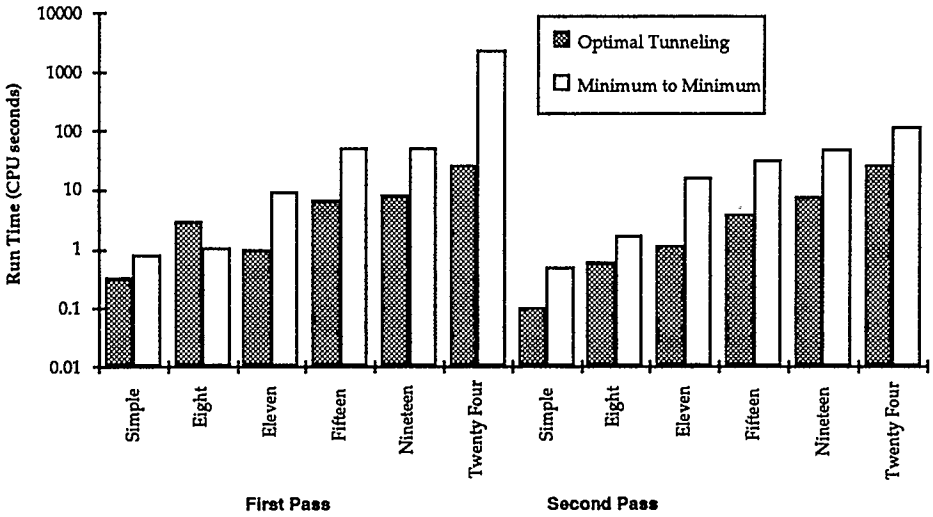


FIGURE 5.10. Run Time For Tile Sliding

5.3.2.3. *Results.* Figure 5.10 shows the run times for the Tile Sliding domain on a logarithmic scale. The only test for which Minimum to Minimum is significantly better is the first attempt at the Eight puzzle. For all subsequent problems, Optimal Tunneling performs much better, up to eighty times faster for the first attempt at the Twenty-four puzzle. Figure 5.11 shows the cumulative run time for the Tile Sliding experiment on a logarithmic scale. Optimal Tunneling shows a clear improvement.

The water pouring analogy in Section 4.2 assumes that the shorter macros proposed by Optimal Tunneling should be applicable more often than Minimum to Minimum macros because they have fewer preconditions. The results in Figure 5.12 support this assumption. This figure shows the number of operators in the operator pool throughout the Tile Sliding experiment. By the end of the first pass through the problems, the Optimal Tunneling heuristic had learned seven macros, and the Minimum to Minimum heuristic had learned twenty-eight. At this stage in the problem, the dynamic filter was invoked, removing four operators from the Minimum to Minimum pool. However, the dynamic filter did not remove any macros from the Optimal

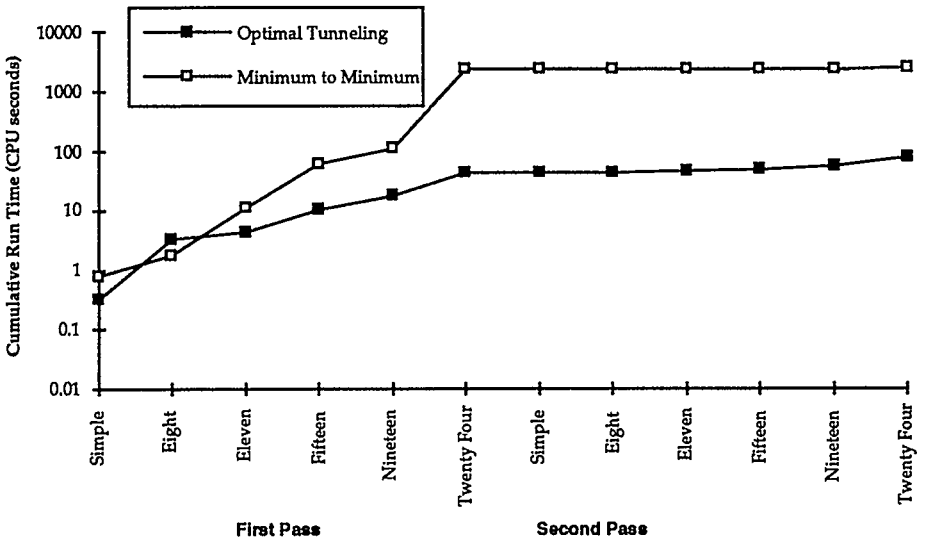


FIGURE 5.11. Cumulative Run Time For Tile Sliding

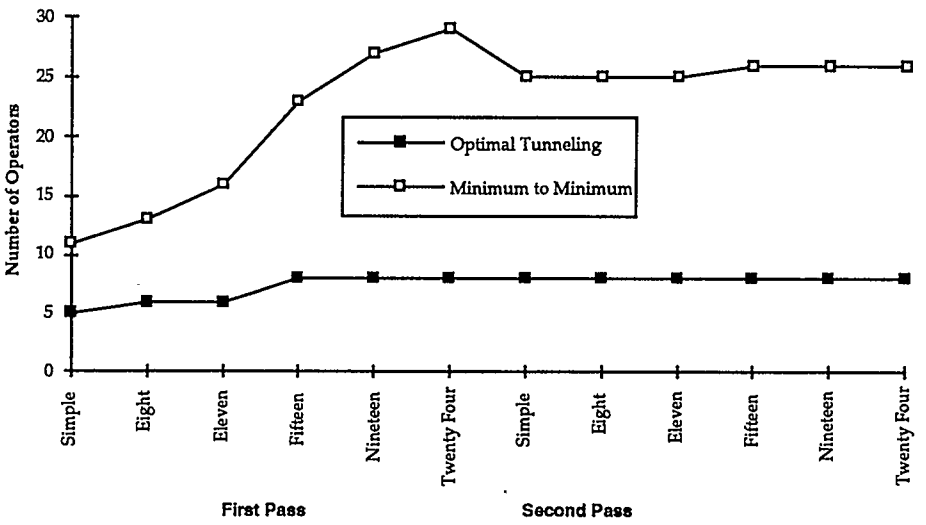


FIGURE 5.12. Number Of Operators For Tile Sliding

Tunneling operator pool. Every macro learned in the first pass was used in later problems. Optimal Tunneling learned fewer but more useful macros. The effect of the Optimal Tunneling macros on the average branching factor is shown in Figure 5.13. The branching factor for Minimum to Minimum is much higher, suggesting that it will have longer run times.

5.3.3. Sokoban. The final domain for testing is the SOKOBAN domain. The problems used in this domain are shown in Figure 5.14. To improve the solution times for these problems (so that the tests could be run in a reasonable time), the problem solver did not consider any states where a ball had been pushed into a corner, since a problem is unsolvable from that point. The heuristic function used for the SOKOBAN domain is given in Section 4.1.1. The limit for the static filter length test was thirty.

Table 5.1 shows the results for the SOKOBAN domain. Optimal Tunneling solved the first screen in less than half the time taken by Minimum to Minimum. Neither

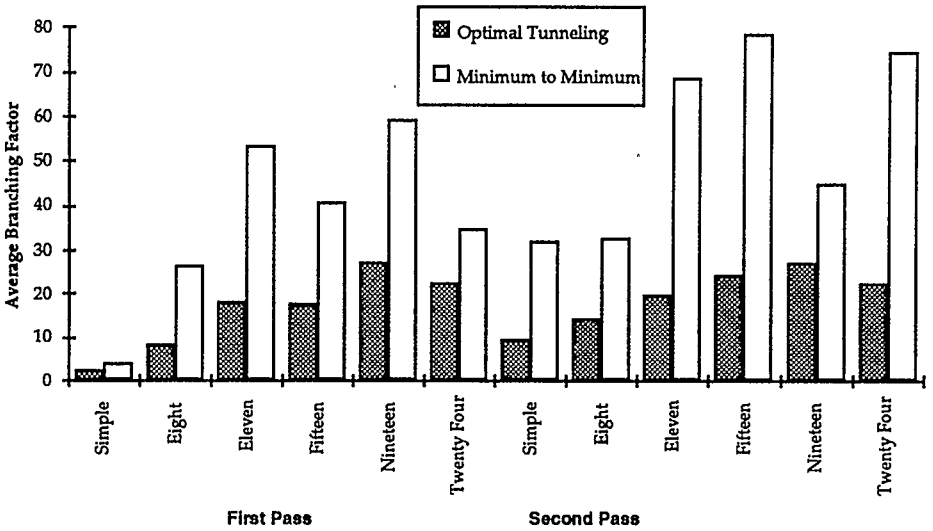
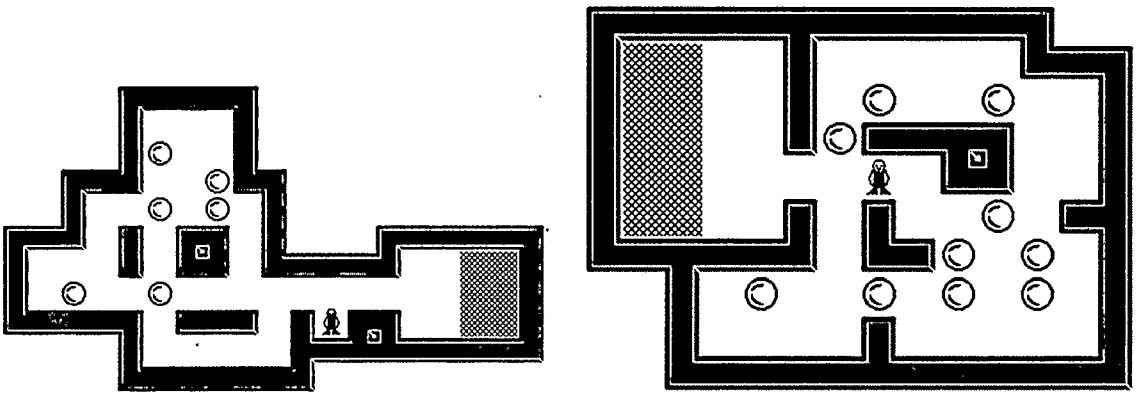


FIGURE 5.13. Average Branching Factor For Tile Sliding



Screen 1 Screen 2

FIGURE 5.14. The Problems Used For The SOKOBAN Experiment

TABLE 5.1. Sokoban Results

Problem	Optimal Tunneling	Minimum to Minimum
Screen 1	2.7 CPU hours	6.7 CPU hours
Screen 2	Unsolved after 24 hours	Unsolved after 24 hours

heuristic solves the problem as quickly as a human can[3] but Table 5.1 still shows the clear superiority of Optimal Tunneling.

5.4. Summary

When each problem was attempted for the first time without cumulative learning across trials, the Optimal Tunneling heuristic resulted in significantly better performance for three of four problems in the Peg Solitaire domain, and performed equally well for the other. For the second trial, with macros learned from the first, Optimal Tunneling resulted in poorer performance on only one problem, but showed superior performance on the others. After dynamic filtering, Minimum to Minimum performed better on only the simplest problem, and Optimal Tunneling was significantly better for the rest.

In the Peg Solitaire domain with cumulative learning across trials , Optimal Tunneling showed a clear improvement in all cases.

With cumulative learning across trials in the Tile Sliding domain, Optimal Tunneling learned fewer and more useful macros, resulting in significantly better performance for all but one problem.

In the SOKOBAN domain too, a problem solving trial using the Optimal Tunneling heuristic took less than half as long as a trial using the Minimum to Minimum heuristic.

These results show that the Optimal Tunneling heuristic results in better performance in many domains, including domains such as Peg Solitaire where the Minimum to Minimum heuristic performs well.

---

[3]About 2 to 3 minutes for someone seeing the problem for the first time.

CHAPTER 6

Concluding Remarks

The results in Chapter 5 replicate Iba's results, confirming that the MACLEARN learning model is effective for solving complex puzzles that cannot be solved in reasonable time with heuristic search alone. The water pouring analogy in Chapter 4 suggests that the Minimum to Minimum heuristic will not work as well as the Optimal Tunneling heuristic. The results of Chapter 5 show that the Optimal Tunneling heuristic out-performs the Minimum to Minimum heuristic, even on problems for which Minimum to Minimum does well.

6.1. Optimal Tunneling

Optimal Tunneling produces shorter, more useful macros than the similar Minimum to Minimum heuristic presented by Iba [13]. Optimal Tunneling is an improvement since its macros:

(1) best reduce search cost

(2) give the most accurate modification to the search space to make the heuristic function correct

(3) result in better performance on comparative tests in the Peg Solitaire, Tile Sliding, and SOKOBAN domains

The water pouring analogy illustrates the effect of macros on the cost of search in problem solving. Optimal Tunneling creates macros that cross exactly the expensive

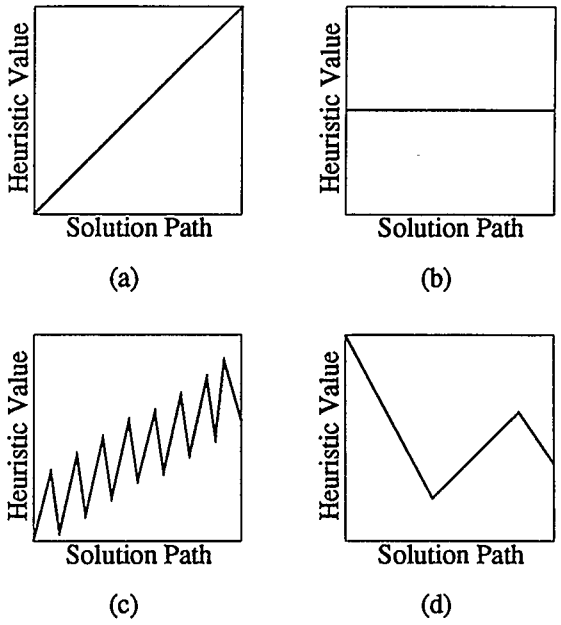


FIGURE 6.1. Some Pathological Heuristic Functions

segment of the heuristic function along the current solution path.

6.2. Future Work

6.2.1. Analysis of Heuristic Functions. The MACLEARN learning model makes some assumptions about the heuristic function it uses. Iba points out that if the evaluation function decreases monotonically, the Minimum to Minimum Heuristic will never be invoked, and no macros will be learned. The same is true of the Optimal Tunneling Heuristic. Some other pathological heuristic functions are shown in Figure 6.1. Figures 6.1(c) and (d) show situations in which the Minimum to Minimum Heuristic will learn macros, but the Optimal Tunneling Heuristic will not. Future work would include a better analysis of how the two systems behave with these pathological heuristic functions, and in what domains they would be common.

6.2.2. Improving the Performance Element. The MACLEARN learning model, even with the Optimal Tunneling heuristic, can solve only the first screen of SOKOBAN. For a domain like Tile Sliding, it is possible to solve difficult problems like the Twenty-four puzzle by learning macros with simple training examples. Although it may be possible to find smaller SOKOBAN problems with which to learn the macros needed to solve more difficult ones, finding out which macros will be needed will not be easy. If the system needs a human guide to find a set of useful macros, then it is not likely to be useful since the human would need to do almost all of the work.

The choice of heuristic search for the performance element causes the most difficulty. Since the search is not explicitly goal-directed, the system often must exhaustively search the nodes in a valley before it can consider new nodes. An example of this problem can be seen in the SOKOBAN domain when a ball has been pushed into a place where it cannot reach the goal. A goal-directed search would discover that the ball could not be moved to the goal, and would backtrack to a state where the ball was free to move to the goal. The heuristic search however, finds that it can continue to improve the heuristic by pushing other balls closer to the goal. It will move all the other balls to as many different positions as it can find before it is forced to consider a state with a higher heuristic value.

One possible solution to these problems is to use a goal-directed search such as that used by GPS [16]. This sort of search would also allow for new types of macro proposers. One possibility is to propose macros when a subgoal has been satisfied.

6.2.3. More Flexible Problem Representation. The MACLEARN system is limited by its choice of problem representation. Although it works well for 2-dimensional search spaces, it is not easily applicable to other domains. For example, it is possible to encode the $2 \times 2 \times 2$ Rubik's cube in the MACLEARN representation

```
        Y Y
        Y Y
R R B B O O W W
R R B B O O W W
        G G
        G G
```

FIGURE 6.2. An Unfolded 2D Representation Of The $2 \times 2 \times 2$ Rubik's Cube

as shown in Figure 6.2, but the operators (Figure 6.3) and macros learned in this domain will not generalize to the $3 \times 3 \times 3$ cube. Also, the symmetry of the domain is lost in the 2D representation.

Other domains that may be suitable to macro learning include symbolic integration and symbolic algebra, but these domains require a more general problem representation.

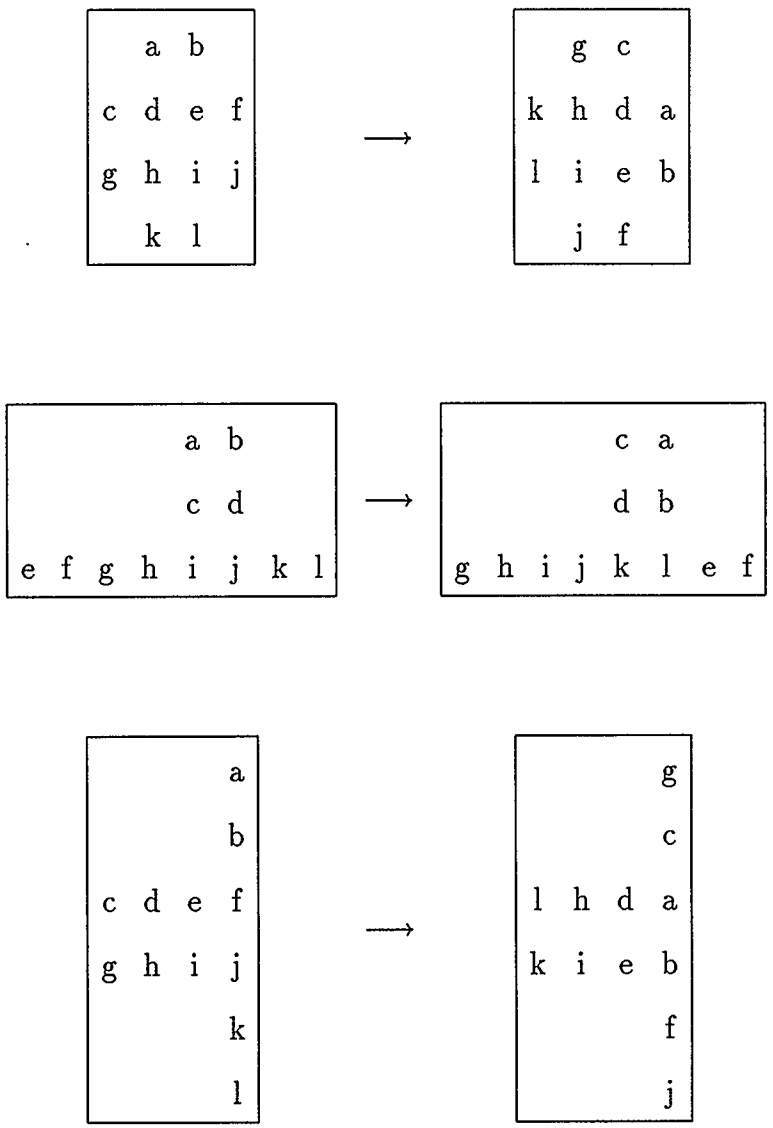


FIGURE 6.3. The Three Operators For The Unfolded $2 \times 2 \times 2$ Rubik's Cube

References

[1] Amihood Amir, Gary Benson, and Martin Farach. Alphabet independant two dimensional matching. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 59–68. The Association For Computing Machinery, May 1992.

[2] Avron Barr and Edward A. Fiegenbaum, editors. *The Handbook of Artificial Intelligence*, volume 1. Addison-Wesley, 1989.

[3] Avron Barr and Edward A. Fiegenbaum, editors. *The Handbook of Artificial Intelligence*, chapter C3b. A*— an optimal search for an optimal solution, pages 64–66. Volume 1 of Barr and Fiegenbaum [2], 1989.

[4] John D. Beasley. *The Ins and Outs of Peg Solitaire.* Recreations in Mathematics. Oxford University Press, 1985.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[6] G.W. Ernst. Sufficient conditions for the success of GPS. *Journal of the ACM*, 16, 1969.

[7] R. E. Fikes, P. Hart, and N. J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 3:251–288, 1972.

[8] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

[9] A.H. Jr. Frey and D. Singmaster. *The Handbook of Cubik Math.* Enslow, Hillside, NJ, 1982.

[10] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, second edition, 1991.

[11] Edward Hordern. *Sliding Piece Puzzles.* Recreations in Mathematics. Oxford University Press, 1986.

[12] G. A. Iba. Leaning by discovering macros in problem solving. In *Proceedings Of the Nineth International Joint Conference on Artifical Intelligence*, pages 640–642, 1985.

[13] G. A. Iba. A heuristic approach to the discovery of macro-operators. *Machine Learning*, 3(4):285–318, 1989.

[14] R. E. Korf. A program that learns to solve Rubik's Cube. In *National Conference on Artificial Intelligence*, pages 164–167, August 1982.

[15] R. E. Korf. Macro-operators: A weak method for learning. *Artificial Intelligence*, 26(1):35–77, 1985.

[16] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice Hall, Englewood Cliffs, NJ, 1972.

[17] I. Pohl. *Machine Intelligence*, chapter Bi-directional search, pages 127–140. American Elsevier, New York, 1971.

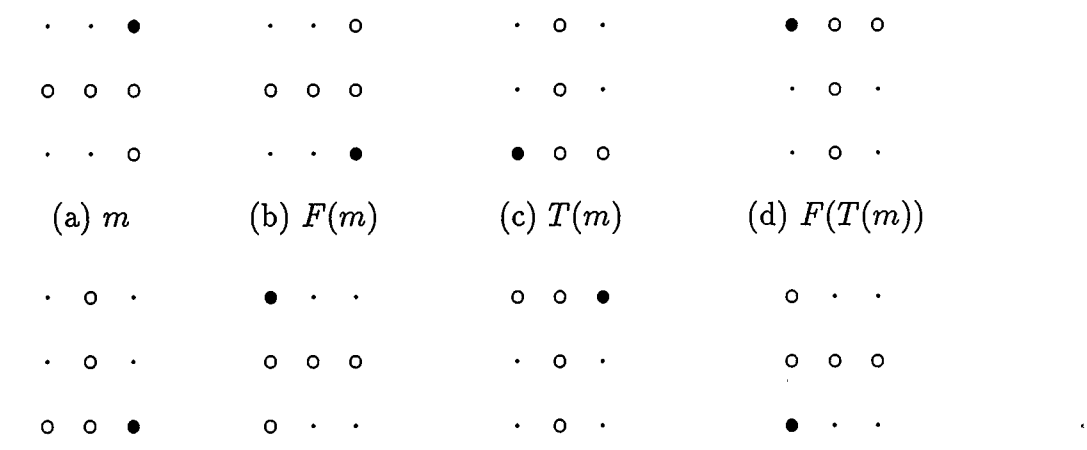
APPENDIX A

Implementation Details

There are several implementation details that can have significant effects on the outcome of a given problem solving trial (Section 5.2). The details of the choices my system makes are given here.

A.1. Creating New Operators

When a new macro is created, or a primitive operator is declared, its reflections and rotations are precomputed based on two operations, reflection about the x axis, and transposition. These operations are chosen because they can be computed efficiently when operators are stored as lists of lists. If we define $F()$ and $T()$ to be



(a) m (b) $F(m)$ (c) $T(m)$ (d) $F(T(m))$

(e) $T(F(m))$ (f) $T(F(T(m)))$ (g) $F(T(F(m)))$ (h) $T(F(T(F(m))))$

FIGURE A.1. The Eight Possible Orientations Of A Macro

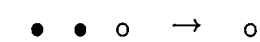


FIGURE A.2. The Primitive Operator For Peg Solitaire

functions that flip and transpose their respective arguments, then there are eight possible orientations of a macro m, as shown in Figure A.1. These orientations are:

$$m, F(m), T(m), F(T(m)), T(F(m)), T(F(T(m))), F(T(F(m))), T(F(T(F(m)))).$$

My system generates these orientations in the order shown above, and duplicates are removed from the list. The new operator is then added to the beginning of the operator list.

Algorithm A.1 describes the method used to expand each state.

ALGORITHM A.1 (EXPAND STATE).
(1) *For each operator in the operator list:*
  (a) *For each pattern in the list of orientations of this operator:*
    (i) *Beginning at the top left corner of the puzzle, attempt to apply the pattern in each possible position. The "before" array of the pattern is tried at each possible location proceeding from left to right, and then moving to the left of the next line.*
    (ii) *Each time a pattern can be applied, look up the resulting state in a hash table of generated states. If it is there, ignore this state, otherwise, compute the heuristic function for the new state, and tag the state with a unique "timestamp."*
    (iii) *Insert the new state into the hash table of generated states, and insert it into the list of unexpanded states, in front of all states with higher heuristic values. If two states have the same heuristic value, the one with the smallest timestamp is placed first.*

Problems such as SOKOBAN are seldom symmetrical, and the orientation of primitive operators is important. The system will eventually generate all possible orientations, but their order depends on which is declared first. My system uses the general rule that movement proceeds from left to right. For example, the primitive operator for Peg Solitaire is declared as in Figure A.2.

Minor changes to these implementation details can have a pronounced effect on the outcome of a given problem trial. To ensure that future researchers can duplicate the results in Chapter 5, the implementation choices are explained in detail here.