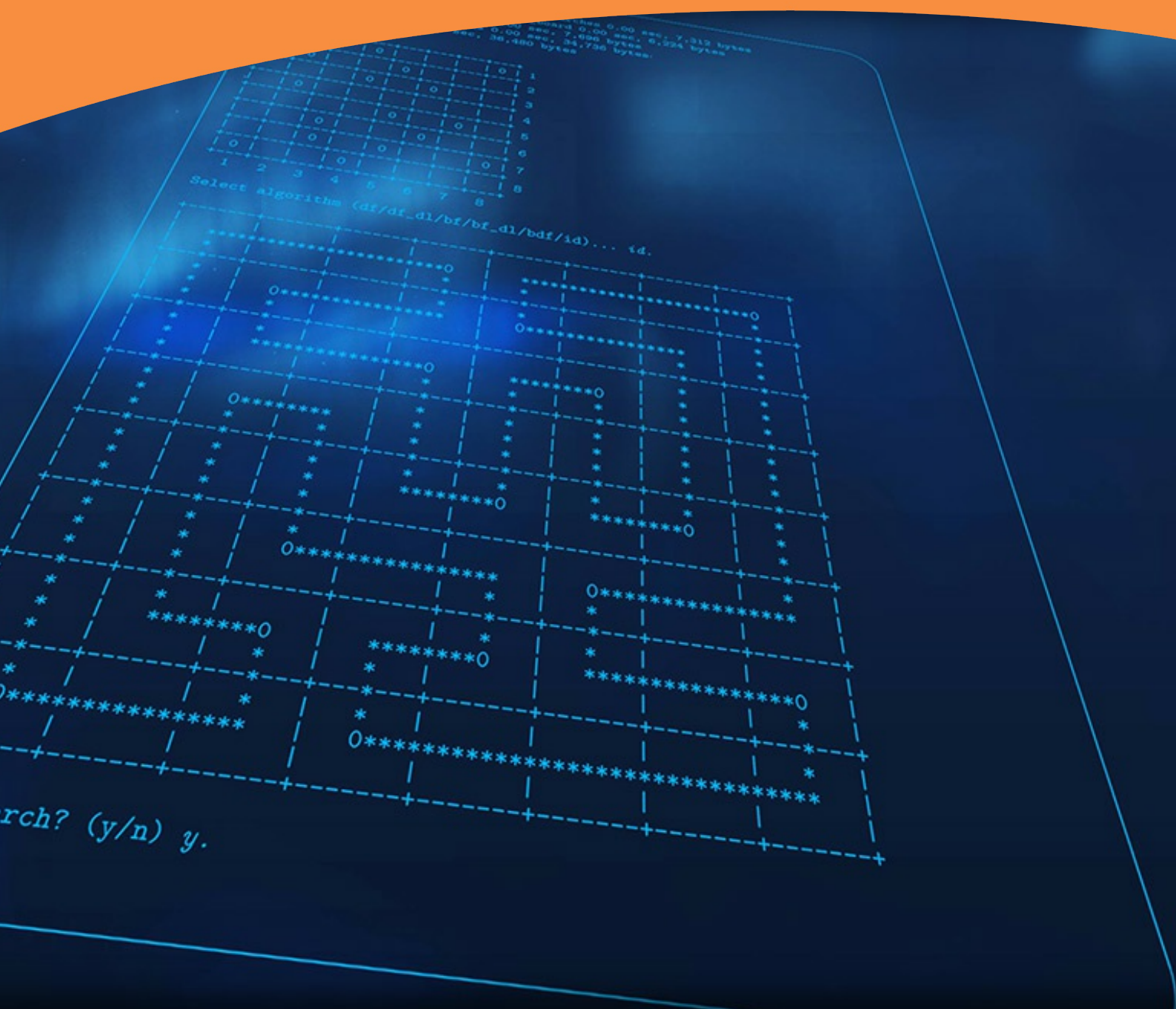


Applications of Prolog

Attila Csenki



Attila Csenki

Applications of Prolog

Applications of Prolog

© 2014 Attila Csenki & bookboon.com

ISBN 978-87-7681-514-1

To my wife Agnes who patiently endured me working on this book for most of my spare time during last two years.

Contents

Preface	15
1 Enigma 1225: Rows are Columns	17
1.1 A Puzzle	17
1.2 First Thoughts	17
1.3 Symbolic Solutions	18
1.4 Implementation Details	20
1.4.1 Design Decisions	20
1.4.2 Admissible Permutations	21
1.4.3 Generating Symbolic Matrices	21
1.4.4 Permuting Rows	22
1.4.5 Transposing	22
1.4.6 Most General Patterned Symbolic Matrices	22
1.4.7 Distinct Rows	24
1.4.8 Evaluating Patterns	25
1.4.9 Computing Totals	28
1.4.10 Complete Implementation	28
1.5 Enhanced Implementation	31
1.5.1 What is Wrong with the Present Implementation?	31
1.5.2 Some Results from the Theory of Permutations	32
1.5.3 Generating Representative Permutations	35
1.5.4 Finishing Touches	43

2	Blind Search	47
2.1	Digression on the Module System in Prolog	47
2.2	Basic Search Problem	49
2.3	Depth First Search	52
2.3.1	Naïve Solution	54
2.3.2	Incremental Development Using an Agenda	54
2.4	Breadth First Search	67
2.5	Bounded Depth First Search	68
2.6	Iterative Deepening	72
2.7	The Module <i>blindsearches</i>	74
2.8	Application: A Loop Puzzle	76
2.8.1	The Puzzle	76
2.8.2	A ‘Hand-Knit’ Solution	77
2.8.3	Project: Automating the Solution Process	83
2.8.4	Project: Displaying the Board	89
2.8.5	Complete Implementation	91
2.8.6	Full Board Coverage	91
2.8.7	Avoiding Multiple Solutions	93
2.8.8	Variants of the Loop Puzzle	95
2.9	Application: The Eight Puzzle	99
2.9.1	The Puzzle	99
2.9.2	Prolog Implementation	100

3	Informed Search	103
3.1	The Network Search Problem with Costs	103
3.1.1	Cost Measures	104
3.1.2	The A -Algorithm	105
3.1.3	Iterative Deepening A^* and its ϵ -Admissible Version	108
3.2	Case Study: The Eight Puzzle Revisited	114
3.2.1	The Heuristics	114
3.2.2	Prolog Implementation	115
3.3	Project: Robot Navigation	118
3.4	Project: The Shortest Route in a Maze	121
3.4.1	Suggested Implementation Details	123
3.5	Project: Moving a Knight	128
4	Text Processing	133
4.1	Text Removal	133
4.1.1	Practical Context	133
4.1.2	Specification	134
4.1.3	Implementation	135
4.1.4	Using a LINUX Shell Script	139
4.1.5	Application: Removing Model Solutions	143
4.2	Text Generation and Drawing with L ^A T _E X	146
4.2.1	Cycloids	146
4.2.2	Task	147
4.2.3	Solution	148
4.3	Exercises	151

A Solutions of Selected Exercises	161
A.1 Chapter 1 Exercises	161
A.2 Chapter 2 Exercises	171
A.3 Chapter 3 Exercises	186
A.4 Chapter 4 Exercises	191
B Software	197
References	199
Index	201
Errata to Volume 1	203

List of Figures

1.1	A Feasible Solution	18
1.2	Hand Computations for Pattern Evaluation	27
1.3	Suggested Hand Computations for <i>total/2</i>	28
1.4	Generating Feasible Solutions by <i>square/5</i>	29
1.5	The Cycles τ_1 and τ_2	32
1.6	Enumeration Scheme for $\{(m, n) : m, n = 0, 1, 2, \dots\}$. (See Exercise 1.12.)	40
1.7	Enumeration Scheme for $\{(m, n) : m, n = 0, 1, 2, \dots\}$. (See Exercise 1.13.)	42
1.8	Suggested Hand Computations for <i>split/4</i>	45
2.1	A Network	48
2.2	The File <i>links.pl</i>	49
2.3	Fragment of the File <i>df1.pl</i>	49
2.4	The Search Tree	50
2.5	The Pruned Search Tree	51
2.6	Depth First Search – The Conduit Model	53
2.7	The File <i>naive.pl</i>	54
2.8	The File <i>df1.pl</i>	56
2.9	Illustrative Query for <i>depth_first/2</i> – First Version	57
2.10	The File <i>df2.pl</i>	58
2.11	Illustrative Query for <i>depth_first/2</i> – Second Version	59
2.12	The New Network Component	59
2.13	Hand Computations for the Query <i>?- depth_first(s, q, Path)</i>	61

2.14	The File <code>df3.pl</code> – Depth First with Closed Nodes and Open Paths	63
2.15	The File <code>df4.pl</code> – Depth First with Path Checking	65
2.16	The File <code>searchinfo.pl</code>	65
2.17	Interactive Session for <i>depth_first/4</i> – Path Checking	66
2.18	A Network (see Exercise 2.4, p. 67)	67
2.19	Breadth First	68
2.20	The File <code>bf.pl</code> – Breadth First with Path Checking	69
2.21	Interactive Session for <i>breadth_first/4</i>	69
2.22	The File <code>bdf.pl</code> – Bounded Depth First (for Exercise 2.7)	72
2.23	The File <code>iterd.pl</code> – Iterative Deepening	73
2.24	Sample Session – Iterative Deepening	74
2.25	Sample Session – Modified Iterative Deepening (for Exercise 2.8)	75
2.26	The File <code>netsearch.pl</code> (for Exercise 2.10)	75
2.27	Sample Session – The Loop Puzzle	78
2.28	The File <code>loop_puzzle1.pl</code>	79
2.29	Constructing a Solution of the Loop Puzzle	80
2.30	The File <code>hand_knit.pl</code>	81
2.31	The File <code>loop_puzzle1a.pl</code>	83
2.32	The File <code>automated.pl</code>	84
2.33	Constructing a Loop	86
2.34	Running the Automated Implementation of the Loop Puzzle	87
2.35	Semi-Automated Solution of the Loop Puzzle	88
2.36	Session for Displaying the Board	89
2.37	Illustrating Exercise 2.16	90
2.38	Illustrating Exercise 2.17	91
2.39	Solving the Puzzle Interactively. (See Exercise 2.18.)	92
2.40	Illustrating Exercise 2.19	93
2.41	Some positions not visited	94

2.42 All positions visited	94
2.43 Solving the Loop Puzzle – Variant One	97
2.44 Solving the Loop Puzzle – Variant Two	98
2.45 An Eight Puzzle	99
2.46 Solving the Eight Puzzle	101
3.1 A Network with Costs	103
3.2 Hand Computations: The Evolution of the Agenda for the <i>A</i> -Algorithm (from <i>d</i> to <i>c</i> in Fig 3.1)	107
3.3 An Interactive Session. (See Exercise 3.1.)	110
3.4 A Directed Network. (See Exercise 3.2.)	111
3.5 Adjacency matrix of the network in Fig. 3.4	111
3.6 Network for Exercise 3.3, Part (c)	113
3.7 Calculating the Manhattan Distance between the tile arrangements in Fig. 2.45	114
3.8 Solving the Eight Puzzle by Heuristic Search	116
3.9 Robot Navigation	119
3.10 Maze Search	122
3.11 Calculating the Euclidean Heuristic H_1	123
3.12 Calculating the Alternative Heuristic H_2	125
3.13 Search Graph for the Gates' Position	126
3.14 Sample Session: Moving a Knight	129
3.15 The Knight Moves One Step	131

4.1	Processing the File <code>exam.tex</code>	134
4.2	The File <code>with_waters</code>	135
4.3	The File <code>without_waters</code>	136
4.4	Running the Shell Script <code>sieve</code>	140
4.5	Another Run of the Shell Script <code>sieve</code>	142
4.6	The File <code>part_sln.tex</code>	143
4.7	Structure of the Printed Exam Script with Solutions	144
4.8	The File <code>part.tex</code>	145
4.9	Running the Shell Script <code>sieve</code>	145
4.10	Drawing a Cycloid ($\phi = \pi/2$)	146
4.11	Prolate Cycloid Drawn with <code>\writecurve</code> from Fig. 4.14 ($r = 5, a = 8, 3.5$ revs)	146
4.12	Curtate Cycloid Drawn with <code>\writecurve</code> similar to Fig. 4.14 ($r = 5, a = 3, 3.5$ revs)	147
4.13	Common Cycloid Drawn with <code>\writecurve</code> similar to Fig. 4.14 ($r = 5, a = 5, 3.5$ revs)	147
4.14	Generating the L ^A T _E X Command <code>\writecurve</code> with <i>define_command/4</i>	148
4.15	‘Quarter’ Cycloid Drawn with <code>\writecurve</code> ($r = 10, a = 4, 1/4$ revs)	151
4.16	Generating the L ^A T _E X Command <code>\defcirc</code> with <i>circ_command/4</i>	151
4.17	Generating the L ^A T _E X Command <code>\defcirc</code> with <i>circ_command/4</i>	152
4.18	Generating the L ^A T _E X Command <code>\defcirc</code> with <i>imp_circ_command/4</i>	153
4.19	Polygon Drawn with <code>\halfcirc</code>	154
4.20	Logarithmic Spiral Drawn with <code>\spiral</code>	157
4.21	Growing Spirals	158
4.22	The File <code>spirals</code>	158
4.23	The File <code>spirals.tex</code>	159
4.24	Running the Shell Script <code>curves</code>	160

A.1	Hand Computations for <i>total/2</i>	164
A.2	Ferrers Diagrams and their Prolog Representations	166
A.3	Creating Distinct Temporary Predicate Names	167
A.4	Annotated Hand Computations for <i>split/4</i>	170
A.5	Hand Computations for the Query <i>?- depth_first(d,c).</i>	172
A.6	Interactive Session for the Query <i>?- depth_first(d,c).</i>	173
A.7	Hand Computations for the Query <i>?- depth_first(u,c).</i>	173
A.8	Tree for Finding Successor Nodes in the New Component	173
A.9	Interactive Session for the Query <i>?- depth_first(u,c).</i>	174
A.10	Sample Session for <i>depth_first/4</i>	176
A.11	Definition of <i>extend_path_dl/3</i>	177
A.12	New Clauses for <i>dfs_loop/4</i>	178
A.13	Updating of the Agenda by <i>dfs_loop/4</i>	178
A.14	Clauses Added to <i>bf.pl</i>	182
A.15	Definition of <i>b_dfs_loop/5</i> (Exercise 2.7)	183
A.16	Modified Version of <i>iterd.pl</i> (Exercise 2.8)	184
A.17	Automated Search	188
A.18	Hand Computations: The Evolution of the Agenda for the <i>A</i> -Algorithm (from node 1 to node 10 in Fig 3.4)	189
A.19	Interactive Session for Searching the Network in Fig. 3.6	190

List of Tables

1.1	CPU times for Various Board Sizes	31
1.2	A Ferrers Diagram	36
1.3	Suggested Examples for Exercise 1.10	38
2.1	CPU Times (in Seconds) for the Eight Puzzle with Blind Search	100
3.1	Straight Line Distances between Nodes in Fig. 3.1	104
3.2	Node Co-ordinates in the Network in Fig. 3.4	110
3.3	Node Co-ordinates in the Network in Fig. 3.6	113
3.4	CPU Times (in Seconds) for the Eight Puzzle with Heuristic Search	117
A.1	Partitions	165
A.2	Example Paths and Prolog Implementations – Case One	175
A.3	Example Paths and Prolog Implementations – Case Two	176
A.4	Values of H	187
A.5	Distances between Nodes (Edge Lengths) in Fig. 3.4	187
A.6	Results for the Eight Puzzle (Hill Climbing and Best First)	190

Preface

This book is the second volume by the author on Prolog programming and its applications written for Ventus. Whereas in the first book [9], specific Prolog programming techniques were explained, in this volume we discuss some areas where Prolog can be fruitfully employed.

Both books owe their existence to the recognition that the higher educational system (in the UK) does not offer enough opportunities for students to experience the satisfaction associated with successfully completing a technical task. In the writer's opinion, the learning experience of today's average student is dominated too much by assessments.

The book comprises four chapters, the first three of them are devoted to Prolog in Artificial Intelligence (AI). The last one is on text processing using Prolog with L^AT_EX in mind.

The first chapter solves an intriguing AI puzzle which was first published in the *New Scientist* magazine [1] in 2003. The Prolog solution presented here combines problem specific knowledge using Finite Mathematics with the well-know AI technique 'generate-and-test'. Even though this chapter did not emanate from my teaching activities, the presentation follows a well-tested pattern: the problem is broken down into manageable and identifiable subproblems which then are more or less readily implemented in Prolog. Many interesting hurdles are identified and solved thereby. The availability of *unification* as a pattern matching tool makes Prolog uniquely suitable for solving such problems. This first chapter is an adaptation of work reported in [7]. Further recent developments on solving this problem can be found in [4].

The second and third chapters are respectively devoted to blind search and informed search. The material presented in them can be used in lectures to teach Prolog for AI as well as in AI lectures themselves. I have tried to compile a varied and interesting mixture of applications most of which won't be available anywhere else. Some of the problems considered here served over the years in my lectures as coursework material, though, for various reasons, the discussion is more thorough here.

The fourth chapter is the least conventional one for a Prolog book. It is in two parts.

1. A tool is developed in Prolog for manipulating L^AT_EX files.
2. Prolog is used to generate L^AT_EX commands for drawing parametric curves in documents written in L^AT_EX.

I also explain here how an SWI-Prolog program can be embedded into a LINUX shell script, removing thereby the need for the user to deal with Prolog directly. This results in applications of direct practical interest.

For the maximum benefit (and fun) readers should work through parts of this book *interactively* with SWI-Prolog. I have tried to retain the experimental and exploratory style of the first volume [9] even though sometimes digression to more theoretical topics was unavoidable.

There are 54 exercises in this book, 32 of them are solved in Appendix A. The last chapter is somewhat of an exception since there the exercises themselves are the main vehicle for conveying the subject material. Therefore, detailed sample solutions are provided for 6 of the 7 exercises in that chapter.

The associated software (Prolog sources, LINUX shell scripts, data files) listed in Appendix B is freely available from the Ventus website. All three systems used here (LINUX, SWI-Prolog, L^AT_EX) are freely available on the Internet.

Bradford,
October 2014

Attila Csenki
a.csenki@bradford.ac.uk

Chapter 1

Enigma 1225: Rows are Columns¹

1.1 A Puzzle

A regular feature in the *New Scientist* magazine is *Enigma*, a weekly puzzle entry which readers are invited to solve. In the 8 February 2003 issue [1] the following puzzle was published.

First, draw a chessboard. Now number the horizontal rows 1, 2, ..., 8, from top to bottom and number the vertical columns 1, 2, ..., 8, from left to right. You have to put a whole number in each of the sixty-four squares, subject to the following:

1. No two rows are exactly the same.
2. Each row is equal to one of the columns, but not to the column with the same number as the row.
3. If N is the largest number you write on the chessboard then you must also write 1, 2, ..., $N - 1$ on the chessboard.

The sum of the sixty-four numbers you write on the chessboard is called your total. What is the largest total you can obtain?

We are going to solve this puzzle here using Prolog. The solution to be described will illustrate two techniques: *unification* and *generate-and-test*.

Unification is a built-in pattern matching mechanism in Prolog which has been used in [9]; for example, the difference list technique essentially depended on it. For our approach here, unification will again be crucial in that the proposed method of solution hinges on the availability of built-in unification. It will be used as a kind of concise symbolic pattern generating facility without which the current approach wouldn't be viable.

Generate-and-test is easily implemented in Prolog. Prolog's backtracking mechanism is used to *generate* candidate solutions to the problem which then are *tested* to see whether certain of the problem-specific constraints are satisfied.

1.2 First Thoughts

Fig. 1.1 shows a board arrangement with all required constraints satisfied. It is seen that the first requirement

¹This chapter is based on [7]. The author thankfully acknowledges the permission by Elsevier to republish the material here.

1	3	1	3	6	6	6	6	6
2	3	3	1	6	6	6	6	6
3	1	3	3	6	6	6	6	6
4	6	6	6	4	5	2	5	4
5	6	6	6	4	4	5	2	5
6	6	6	6	5	4	4	5	2
7	6	6	6	2	5	4	4	5
8	6	6	6	5	2	5	4	4
	1	2	3	4	5	6	7	8

Figure 1.1: A Feasible Solution

is satisfied since the rows are all distinct. The second condition is also seen to hold whereby rows and columns are interrelated in the following fashion:

Column	1	2	3	4	5	6	7	8
Row	2	3	1	5	6	7	8	4

We use the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 3 & 1 & 5 & 6 & 7 & 8 & 4 \end{pmatrix} \quad (1.1)$$

to denote the corresponding column-to-row transformation. The board also satisfies the latter part of the second condition since no row is mapped to a column in the same position. In terms of permutations, this requirement implies that no entry remains fixed; these are those permutations which in our context are *permissible*.² The third condition is obviously also satisfied with $N = 6$. The board's total is 301, not the maximum, which, as we shall see later, is 544.

1.3 Symbolic Solutions

The solution scheme described below in i-v is based on first generating all feasible solutions (an example of which was seen in Sect. 1.2) and then choosing a one with the maximum total.

- i. Take an admissible permutation, such as π in (1.1).
- ii. Find an 8×8 matrix with *symbolic* entries whose rows and columns are interrelated by the permutation

²Such permutations are called *derangements* ([3], p. 73).

in i. As an example, let us consider for the permutation π two such matrices, $\mathbf{M}_1$ and $\mathbf{M}_2$, with

$$\mathbf{M}_1 = \begin{bmatrix} X_3 & X_1 & X_3 & X_6 & X_6 & X_6 & X_6 & X_6 \\ X_3 & X_3 & X_1 & X_6 & X_6 & X_6 & X_6 & X_6 \\ X_1 & X_3 & X_3 & X_6 & X_6 & X_6 & X_6 & X_6 \\ X_6 & X_6 & X_6 & X_4 & X_5 & X_2 & X_5 & X_4 \\ X_6 & X_6 & X_6 & X_4 & X_4 & X_5 & X_2 & X_5 \\ X_6 & X_6 & X_6 & X_5 & X_4 & X_4 & X_5 & X_2 \\ X_6 & X_6 & X_6 & X_2 & X_5 & X_4 & X_4 & X_5 \\ X_6 & X_6 & X_6 & X_5 & X_2 & X_5 & X_4 & X_4 \end{bmatrix}$$

$$\mathbf{M}_2 = \begin{bmatrix} Y_3 & Y_1 & Y_3 & Y_1 & Y_1 & Y_1 & Y_1 & Y_1 \\ Y_3 & Y_3 & Y_1 & Y_1 & Y_1 & Y_1 & Y_1 & Y_1 \\ Y_1 & Y_3 & Y_3 & Y_1 & Y_1 & Y_1 & Y_1 & Y_1 \\ Y_1 & Y_1 & Y_1 & Y_4 & Y_5 & Y_2 & Y_5 & Y_4 \\ Y_1 & Y_1 & Y_1 & Y_4 & Y_4 & Y_5 & Y_2 & Y_5 \\ Y_1 & Y_1 & Y_1 & Y_5 & Y_4 & Y_4 & Y_5 & Y_2 \\ Y_1 & Y_1 & Y_1 & Y_2 & Y_5 & Y_4 & Y_4 & Y_5 \\ Y_1 & Y_1 & Y_1 & Y_5 & Y_2 & Y_5 & Y_4 & Y_4 \end{bmatrix}$$

$\mathbf{M}_1$ and $\mathbf{M}_2$ both satisfy conditions 1 and 2. We also observe that the pattern of $\mathbf{M}_2$ may be obtained from that of $\mathbf{M}_1$ by specialization (by matching the variables X_1 and X_6). Thus, any total achievable for $\mathbf{M}_2$ is also achievable for $\mathbf{M}_1$. For any given permissible permutation, we can therefore concentrate on the *most general* pattern of variables, $\mathbf{M}$. (We term a pattern of variables *most general* if it cannot be obtained by specialization from a more general one.) All this is reminiscent of ‘unification’ and the ‘most general unifier’, and we will indeed be using Prolog’s unification mechanism in this step.

iii. Verify condition 1 for the symbolic matrix $\mathbf{M}$.³ Once this test is passed, we are sure that also the latter part of condition 2 is satisfied.⁴

iv. We now *evaluate* the pattern $\mathbf{M}$. If N symbols have been used in $\mathbf{M}$, assign the values $1, \dots, N$ to them

³This test is necessary since at this stage a matrix may have been generated failing to satisfy condition 1 as is illustrated by the (admissible) permutation

$$\rho = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 3 & 1 & 5 & 4 & 7 & 8 & 6 \end{pmatrix} \quad (1.2)$$

and the corresponding most general matrix $\mathbf{M}_3$:

$$\mathbf{M}_3 = \begin{bmatrix} Z_4 & Z_1 & Z_4 & Z_9 & Z_9 & Z_5 & Z_6 & Z_7 \\ Z_4 & Z_4 & Z_1 & Z_9 & Z_9 & Z_7 & Z_5 & Z_6 \\ Z_1 & Z_4 & Z_4 & Z_9 & Z_9 & Z_6 & Z_7 & Z_5 \\ Z_9 & Z_9 & Z_9 & Z_3 & Z_3 & Z_{10} & Z_{10} & Z_{10} \\ Z_9 & Z_9 & Z_9 & Z_3 & Z_3 & Z_{10} & Z_{10} & Z_{10} \\ Z_7 & Z_6 & Z_5 & Z_{10} & Z_{10} & Z_8 & Z_2 & Z_8 \\ Z_5 & Z_7 & Z_6 & Z_{10} & Z_{10} & Z_8 & Z_8 & Z_2 \\ Z_6 & Z_5 & Z_7 & Z_{10} & Z_{10} & Z_2 & Z_8 & Z_8 \end{bmatrix}$$

⁴Were it not so, there would exist a row and a column with the same index such that the two were identical. However, this row will be identical (by way of the *admissible* permutation) to some other column too. Hence two columns and therefore also two rows would be identical, thus failing the test.

in reverse order by first assigning N to the most frequently occurring symbol, $N - 1$ to the second most frequently occurring symbol etc. The total thus achieved will be a maximum for the given pattern $\mathbf{M}$.

- v. The problem is finally solved by generating and evaluating all patterns according to i–iv and selecting a one with the maximum total.

1.4 Implementation Details

1.4.1 Design Decisions

The original formulation from the *New Scientist* uses a chessboard but the problem can be equally set with a square board of any size. In our implementation, we shall allow for any board size since this will allow the limitations of the method employed to be explored.

We write matrices in Prolog as lists of their rows which themselves are lists. Permutations will be represented by the list of the bottom entries of their two-line representation; thus, $[2, 3, 1, 5, 6, 7, 8, 4]$ stands for π in (1.1).

1.4.2 Admissible Permutations

First, we want to generate all permutations of a list. Let us assume that we want to do this by the predicate *permute(+List,-Perm)* and let us see how *List* = [1, 2, 3, 4] might be permuted. A permuted list, *Perm* = [3, 4, 1, 2] say, may be obtained by

- Removing from *List* the entry *E* = 3, leaving the reduced list *R* = [1, 2, 4]
- Permuting the reduced list *R* to get *P* = [4, 1, 2]
- Assembling the permuted list as *[E|P]* = [3, 4, 1, 2].

Lists with a single entry are left unchanged. This gives rise to the definition

```
permute([X],[X]).
permute(L,[E|P]) :- remove_one(L,E,R), permute(R,P).
```

with the predicate *remove_one(+List,?Entry,?Reduced)* defined by

```
remove_one([H|T],H,T).
remove_one([H|T],E,[H|L]) :- remove_one(T,E,L).
```

(Here we remove either the head or an entry from the tail.) For a permutation to be admissible, all entries must have changed position. We implement this by

```
admissible(L,P) :- permute(L,P), all_changed(L,P).

all_changed([X],[Y]) :- X \= Y.
all_changed([H1|T1],[H2|T2]) :- H1 \= H2, all_changed(T1,T2).
```

Exercise 1.1. Provide an alternative definition of *remove_one/3* by using one clause and *append/3*. ■

1.4.3 Generating Symbolic Matrices

To generate a list of *N* unbound variables, *L*, we use *var_list(+N,-L)* which is defined in terms of *length(-L,+N)* by

```
var_list(N,L) :- length(L,N).
```

(See [9, p. 110, footnote 15].) Matrices with distinct symbolic entries may now be produced by mapping; for example, a 3×2 matrix is obtained by

```
?- maplist(var_list,[2,2,2],M).
M = [[_G370, _G373], [_G379, _G382], [_G388, _G391]]
```

Exercise 1.2. Use the above idea to define *var_matrix(+Size,-M)* for generating a square symbolic matrix of any size. ■

1.4.4 Permuting Rows

This is accomplished by `list_permute(+Perm,+L,-P)` as indicated below.

```
?- var_matrix(3,M), list_permute([3,1,2],_M,_P),
   write_matrix(_M), nl, write_matrix(_P).
[_G779, _G782, _G785]
[_G791, _G794, _G797]
[_G803, _G806, _G809]

[_G803, _G806, _G809]
[_G779, _G782, _G785]
[_G791, _G794, _G797]
```

(The permutation *Perm* establishes a correspondence between the entries of *P* and those of *L*.)

Exercise 1.3. Define the predicate `list_permute/3` by recursion, using `nth1/3` from [9, p. 107]. ■

1.4.5 Transposing

This will be accomplished by `transpose(+M,-T)`.

```
?- maplist(var_list,[2,2,2],_M), transpose(_M,_T),
   write_matrix(_M), nl, write_matrix(_T).
[_G779, _G782]
[_G788, _G791]
[_G797, _G800]

[_G779, _G788, _G797]
[_G782, _G791, _G800]
```

Exercise 1.4. Use `maplist/3` to define `transpose/2`. Allow for any not necessarily square matrix as indicated above.

Hint. First define a predicate `col(+Matrix,+N,-Column)` for returning the *N*th column of a matrix. ■

1.4.6 Most General Patterned Symbolic Matrices

It is now that Prolog shows its true strength: we use *unification* to generate symbolic square matrices with certain patterns.⁵ For example, we may produce a 3×3 symmetric matrix thus

```
?- var_matrix(3,M), transpose(_M,_M), write_matrix(_M).
[_G535, _G538, _G541]
[_G538, _G550, _G553]
[_G541, _G553, _G565]
```

⁵Trying to produce the results in this section by a programming language without built-in unification will be a much more involved exercise.

More importantly, we are now in a position to produce symbolic matrices with prescribed patterns. For example, below we generate the most general 3×3 matrix whose rows and columns are interrelated by the permutation

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

```
?- var_matrix(3,_M), list_permute([3,1,2],_M,_P),
   transpose(_P,_M), write_matrix(_M).
[_G748, _G748, _G754]
[_G754, _G748, _G748]
[_G748, _G754, _G748]
```

Unification is again seen to play a crucial rôle here as *_M* is *declared* to be the transpose of *_P*:

- *transpose/2* receives in its first argument the Prolog term for *_P*.
- The term for the transpose of *_P* is returned in the second argument of *transpose/2*.
- This then is unified with the term for *_M* thereby producing the intended pattern.

1.4.7 Distinct Rows

We want to test whether all rows of a matrix with symbolic entries are distinct. Matrices are lists, we therefore need to test for distinctness of list entries which are Prolog *terms*. The matrix $[[A, B], [C, D]]$ should pass the test, whereas $[[A, B], [A, B]]$ should not. The negation of the unification operator ($\neq/2$) cannot tell apart the rows of the first matrix; we need here a ‘stronger’ (i. e. more specialized) notion of equality as defined by the *term equivalence* operator $==/2$ and its negation, $\neq/2$. (See inset overleaf.) Thus, using $\neq/2$ will allow the rows of the former matrix to be recognized as different, whereas those of the latter are verified identical.

```
?- [A, B] \neq [C, D].
A = _G240
B = _G243
C = _G246
D = _G249
Yes
?- [A, B] \neq [A, B].
No
```

Built-in Predicates: $==/2$ and $\neq/2$

These two predicates are used to test for term ‘equivalence’ and its negation, respectively. Two terms are equivalent if there exists a term to which both of them have been bound *prior to* the invocation of $==/2$. For example, the query

```
?- X = u, g(X,V) = Y, f(h(g(u,V)),Y) == f(h(Y),g(X,V)).
X = u
V = _G448
Y = g(u, _G448)
Yes
```

succeeds since both sides have been bound (by prior unification) to the term $f(h(g(u,V)),g(u,V))$. However, the query

```
?- f(h(g(u,V)),Y) == f(h(Y),g(X,V)).
No
```

fails even though the two terms are unifiable:

```
?- f(h(g(u,V)),Y) = f(h(Y),g(X,V)).
V = _G325
Y = g(u, _G325)
X = u
Yes
```

Exercise 1.5. Use $\neq/2$ to define a predicate *distinct/1* for testing the distinctness of entries of a list as discussed above. ■

1.4.8 Evaluating Patterns

Given a patterned symbolic matrix, we want to sort the list of its entries according to their frequencies of occurrence and assign the rank order to each. For example, in the matrix *M* from the second query in Sect. 1.4.6, p. 22, the entry *_G748* occurs six times while *_G754* occurs thrice. Therefore, as shown below, *_G754* and *_G748* will be assigned the values 1 and 2 respectively.

```
?- var_matrix(3,M), list_permute([3,1,2],M,P),
   transpose(_P,M), eval_matrix(M,Freq), write_matrix(_M).
[2, 2, 1]
[1, 2, 2]
[2, 1, 2]
```

```
Freq = [ (3, 1), (6, 2)]
```

This shall be accomplished by the predicate *eval_matrix(?M,-Freq)*; it expects a symbolic matrix *M* in its first argument which then is unified with an integer matrix whose each entry will be the rank order of the frequency of the corresponding symbolic entry. The second argument *Freq* is unified with the list of frequencies for each number in the matrix as indicated above.

The hand computations in Fig. 1.2 on p. 27 indicate the steps involved in implementing *eval_matrix/2*.

- ① Produce the list of matrix entries by *flatten(+Matrix,-Entries)*.
- ② Discard multiple occurrences by *setof(E,member(E,+Entries),-Set)*.
- ③ Use *maplist(count_var(+Entries),+Set,-Multiplicities)* to count how many times each variable occurs in the matrix.

Exercise 1.6. Define the predicate *count_var(+VarList,+Var,-Num)*. It will behave as follows.

```
?- count_var([_A,_B,_A,_C,_B,_A],_B,N).
N = 2
```

■

- ④ Use *zip(+Multiplicities,+Set,-Frequencies)* to obtain the list of matrix entry frequencies by *zipping* the lists produced in ② and ③.

Exercise 1.7. Define the predicate *zip/3*. It should behave as follows.

```
?- zip([1,2,3],[a,b,c],L).
L = [ (1, a), (2, b), (3, c)]
```

■

- ⑤ Use *sort(+Frequencies,-FreqSorted)* (Prolog's built-in *sort/2*) to sort the pairs from ④. Tuples with less frequent matrix entries will precede those with more frequent ones.
- ⑥ Use *maplist(snd,+FreqSorted,-VarsSorted)* to retain the tuples' second entries only. We get a complete list of matrix entries, with no multiple copies, featuring in the rank order of their frequencies. *snd/2* extracts the second entry of a 2-tuple and is defined by

`snd( (_,X),X).`

- ⑦ Use `length(+VarsSorted,-NVars)` to count the number of distinct matrix entries.
- ⑧ Use `from_to/3` to generate the list of integers `[1, ..., NVars]`. (The predicate `from_to/3` is known from [9, p. 17].)
- ⑨ Unify each variable in `VarsSorted` with the rank order of its frequency. A single call to `from_to(1,+NVars,?VarsSorted)` will accomplish both steps, ⑧ and ⑨. The effect of this call will also be that
 - The initial (input) matrix will be bound to the integer matrix of frequency ranks. This will form the first output of `eval_matrix/2`.
 - `FreqSorted` will be bound to the list of frequency pairs, forming the second output of `eval_matrix/2`.

The complete definition of `eval_matrix/2`, now a mere sequencing of clauses from ①–⑨, will be found in the source file `enigma.pl`.

The predicate *eval_matrix/2* has been defined in a style reminiscent of that used in *functional programming*. (The predicates *maplist/3*, *zip/3* and *snd/2* have indeed direct analogues in Haskell [30].) In [24], Parker espouses the virtues of this style for Prolog and calls it the ‘stream data analysis paradigm’. Fig. 1.2 corresponds to what is called in [24] a ‘dataflow diagram’ or ‘Henderson diagram’.

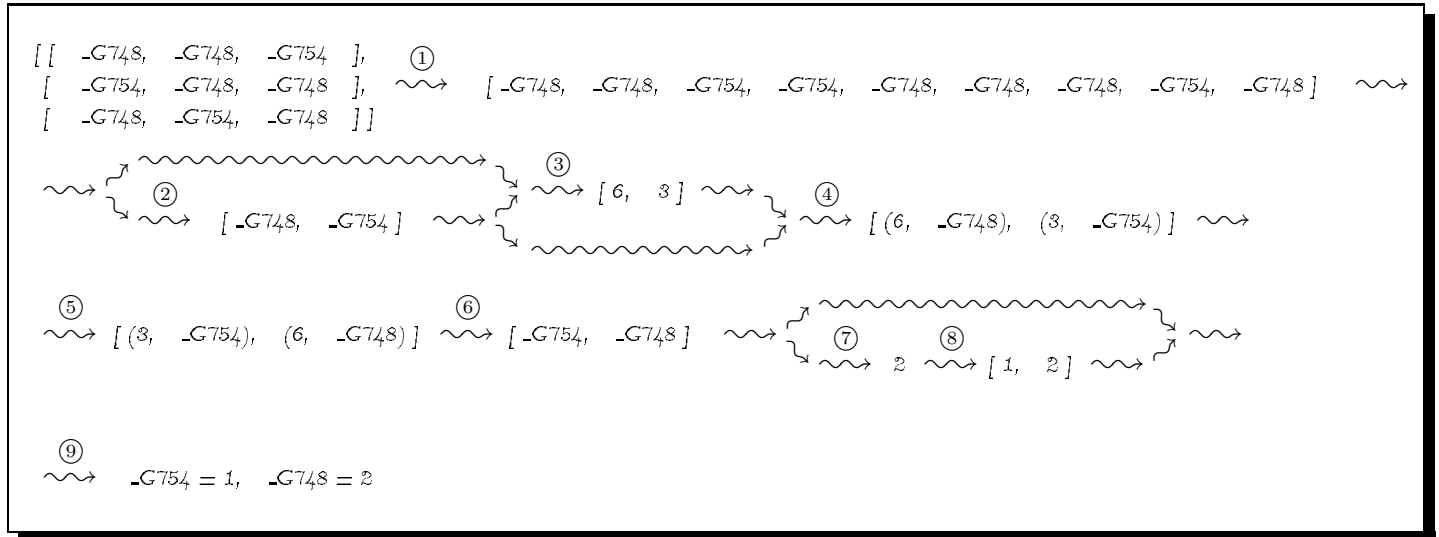


Figure 1.2: Hand Computations for Pattern Evaluation

```

total( [(1,10),(2,100),(3,1000)], Total) ~~~>
total([(1,10),(2,100),(3,1000)], 0, Total) ~~~>
total([(2,100),(3,1000)], 10, Total) ~~~>
total([(3,1000)], 210, Total) ~~~> total([], 3210, Total) ~~~>
Total = 3210 ~~~> success

```

Figure 1.3: Suggested Hand Computations for *total/2*

1.4.9 Computing Totals

Exercise 1.8. For the computation of the matrix total we shall need a predicate *total(+IntPairs,-Total)* which should sum the product of paired entries as exemplified below.

```

?- total([(1,10),(2,100),(3,1000)],Total).
Total = 3210

```

Define *total/2* by the accumulator technique along the hand computations shown in Fig. 1.3. ■

1.4.10 Complete Implementation

In (P-1.1), we show the definition of *square/5* which has been assembled from the predicates in Sects. 1.4.2–1.4.9.

Prolog Code P-1.1: Definition of *square/5*

```

1 square(Size,M,Total,Freq,Perm) :- var_matrix(Size,M),
2                                   from_to(1,Size,One_to_Size),
3                                   admissible(One_to_Size,Perm),
4                                   list_permute(Perm,M,P),
5                                   transpose(P,M),
6                                   distinct(M),
7                                   eval_matrix(M,Freq),
8                                   total(Freq,Total).

```

square/5 may be used to search for feasible solutions as shown by the query in Fig. 1.4 for a 4×4 board. We know that all boards with the maximum total will be amongst those generated by the current process. Therefore, the largest of all totals thus generated will be the maximum total. We use *setof/3* to obtain the sorted list of all totals generated (without duplicates) and select the maximum value by the built-in predicate *last/2*:⁶

⁶There is some inconsistency between versions of SWI-Prolog here. Version 3.4.5 is used in the query below, but, the order of the arguments in *last/2* will have to be reversed if using version 5.2.7.

```
?- square(4,_M,Total,Freq,Perm), write_matrix(_M).  
[1, 1, 2, 3]  
[1, 1, 3, 2]  
[3, 2, 4, 4]  
[2, 3, 4, 4]  
  
Total = 40  
Freq = [ (4, 1), (4, 2), (4, 3), (4, 4)]  
Perm = [2, 1, 4, 3] ;  
[1, 2, 2, 1]  
[1, 1, 2, 2]  
[2, 1, 1, 2]  
[2, 2, 1, 1]  
  
Total = 24  
Freq = [ (8, 1), (8, 2)]  
Perm = [2, 3, 4, 1] ;  
...
```

Figure 1.4: Generating Feasible Solutions by *square/5*

```
?- setof(_Tot,_M^_Freq^_Perm^square(8,_M,_Tot,_Freq,_Perm),Tots),
    last(Max,Tots).
Tots = [160, 244, 288, 301, 400, 544]
Max = 544
```

We now know that the maximum total is 544 and may find a board with that total (and the corresponding permutation) by

```
?- square(8,_M,544,_,Perm), write_matrix(_M).
[ 1  1  2  3  4  5  6  7]
[ 1  1  3  2  5  4  7  6]
[ 3  2  8  8  9 10 11 12]
[ 2  3  8  8 10  9 12 11]
[ 5  4 10  9 13 13 14 15]
[ 4  5  9 10 13 13 15 14]
[ 7  6 12 11 15 14 16 16]
[ 6  7 11 12 14 15 16 16]
```

```
Perm = [2, 1, 4, 3, 6, 5, 8, 7]
```

Exercise 1.9. Define the predicate *write_matrix/1* for displaying on the terminal an *integer* matrix with non-negative entries, right justified. In your definition, you should use *writeln(+Format,+Arguments)* (Prolog's formatted *write*); see inset. The built-in predicates *concat_atom/2* [9, p. 126] and *int_to_atom/2* (see inset) may be used to construct *writeln*'s first argument. ■

Built-in Predicate: *writeln(+Format,+Arguments)*

This is one of Prolog's predicates for formatted *write*. *Arguments* is a list whose entries are displayed on the terminal according to the atom *Format*. Example:

```
?- writeln('[%8r%8r%8r]',[12, 345, 6789]).
[      12      345     6789]
```

displays the list *[12, 345, 6789]* with its entries right justified, each occupying up to eight digits. Consult the manual [33] for the options available for *Format*.

Built-in Predicate: *int_to_atom(+Int,-Atom)*

Unifies *Atom* with the ASCII representation of *Int*. Example:

```
?- int_to_atom(1953,A).
A = '1953'
```

<i>Size</i>	3	4	5	6	7	8	9
<i>CPU Seconds</i>	0.00	0.06	0.11	2.03	15.37	209.59	3,334.14

Table 1.1: CPU times for Various Board Sizes

1.5 Enhanced Implementation

1.5.1 What is Wrong with the Present Implementation?

The implementation obtained in Sect. 1.4.10 has serious limitations. Table 1.1 shows the CPU times needed for solving the puzzle for up to size 9 on a 300 MHz PC. The size of the original puzzle seems to be the practical limit of what can be solved by this method.⁷ Table 1.1 indicates that the computing time increases roughly with the factorial of *Size*. This means for the original puzzle that $8! = 40,320$ permutations have to be generated of which 14,833 will be admissible.⁸ Each of these will give rise to a patterned symbolic matrix, each to be tested by *distinct/1*. The number of patterned matrices passing this test is 13,713.⁹ All of them are then evaluated, resulting in a list with 13,713 entries. After removing duplicates with *setof/3*, we end up with a list of just six values!

There is obviously a great deal of duplication of effort here.

To reduce the number of permutations to be considered, we are going to introduce in the next section a *partitioning* of the set of all permutations into subsets, called *types*, such that permutations of the same type will *share* certain pertinent properties. More precisely, each of the following properties will be such that permutations *of the same type* either all have it or none has it.¹⁰

- Being admissible,
- For admissible permutations, the corresponding most general symbolic pattern having distinct rows.

Furthermore,

- For permutations of the same type, the corresponding most general symbolic pattern will evaluate to the same maximum total.

⁷There is another problem for larger sizes which could be overcome, however. For sizes exceeding 9, insufficient memory will be available for using *setof/3* to collect the values of total. To remedy the situation, we could instead calculate the maximum total in an incremental fashion by using, for instance, *assert/1* to save in the database the most recent maximum value of total.

⁸The number of admissible permutations can be found by the query

```
?- bagof(_A, admissible([1,2,3,4,5,6,7,8],_A), _As), length(_As,L).
L = 14833
```

Alternatively, the number of admissible permutations of $\{1, \dots, n\}$, a_n , may be calculated by the recurrence relation

$$a_n = n! - (f_{1n} + f_{2n} + \dots + f_{(n-1)n} + 1)$$

where

$$f_{in} = \binom{n}{i} a_{n-i}$$

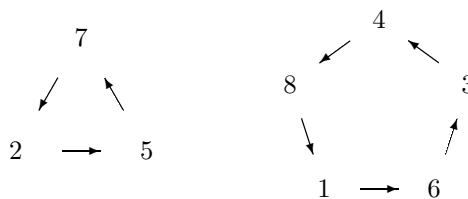
denotes the number of permutations of $\{1, \dots, n\}$ which leave *exactly* i entries fixed. Start with $a_1 = 0$. Other ways of calculating a_n may be found in [3, p. 73].

⁹We find this by the query

```
?- bagof(_Tot, _M^_Freq^_Perm^square(8,_M,_Tot,_Freq,_Perm), _Tots), length(_Tots,L).
L = 13713
```

The matrix M_3 in footnote 3, p. 19, is an example for a pattern which will be tested by *distinct/1* and fail.

¹⁰We may call them therefore *type-properties*.

Figure 1.5: The Cycles τ_1 and τ_2

It will therefore suffice to concentrate on a *representative permutation from each type* (Sect. 1.5.3). Before elaborating on this idea, however, we first review some results from the Theory of Permutations [3].

1.5.2 Some Results from the Theory of Permutations

The Cycle Notation for Permutations

Let us look at the permutation

$$\tau = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 6 & 5 & 4 & 8 & 7 & 3 & 2 & 1 \end{pmatrix}$$

It can be thought of as the composition of two *cycles* τ_1 and τ_2 with

$$\tau_1 = \begin{pmatrix} 2 & 5 & 7 \\ 5 & 7 & 2 \end{pmatrix}, \quad \tau_2 = \begin{pmatrix} 1 & 3 & 4 & 6 & 8 \\ 6 & 4 & 8 & 3 & 1 \end{pmatrix}$$

It is seen from Fig. 1.5 that both cycles (as the name implies) effect a *cyclical* interchange on a subset of $\{1, \dots, 8\}$; these subsets form a *partition* of $\{1, \dots, 8\} = \{2, 5, 7\} \cup \{1, 3, 4, 6, 8\}$. We may use the *cycle notation* to denote cycles: $\tau_1 = (5 \ 7 \ 2)$, $\tau_2 = (6 \ 3 \ 4 \ 8 \ 1)$. The permutation τ is said to be the *product* of the cycles τ_1 and τ_2 ,

$$\tau = (5 \ 7 \ 2)(6 \ 3 \ 4 \ 8 \ 1) \quad (1.3)$$

As the individual cycles of a product operate on disjoint sets, the order in which the cycles are listed is immaterial, though shorter cycles are usually written before longer ones. Thus $\tau = (6 \ 3 \ 4 \ 8 \ 1)(5 \ 7 \ 2)$. The entries of a cycle in the cycle notation may be *rotated* [9]; for example, $(3 \ 4 \ 8 \ 1 \ 6)$ still refers to the cycle τ_2 .

Another example of a permutation in the cycle notation is

$$\rho = (4 \ 5)(1 \ 2 \ 3)(6 \ 7 \ 8) \quad (1.4)$$

from (1.2) on p. 19; it is the product of three cycles.

Finally, permissible permutations (so-called *derangements*) are now easily recognized as those without a 1-cycle.

Types

The permutation τ in (1.3) is the product of two cycles, τ_1 and τ_2 , of length 3 and 5, respectively. Therefore, τ is said to be of *type* $[3^1 5^1]$.¹¹ π in (1.1) is another permutation of the same type, since

$$\pi = (3\ 1\ 2)(7\ 8\ 4\ 5\ 6) \quad (1.5)$$

On the other hand, ρ in (1.4) is seen to be of type $[2^1 3^2]$.

We note in passing that each type corresponds to a *partition* of the number of elements permuted. A partition of a positive whole number is its representation as the sum of some positive whole numbers. For example, the above types define the partitions $8 = 3 + 5$ and $8 = 2 + 3 + 3$.

Types in our context become significant by the following

Observation. Column-to-row transformations of the same type give rise to most general patterned symbolic matrices which are essentially the same in that they can be transformed into each other by appropriate row-to-row and column-to-column rearrangements.

We won't prove this result here but illustrate it by an example. To determine the most general symbolic matrix for τ from that of π , proceed as follows.

1. Write the permutations π and τ in cycle notation (as in (1.5) and (1.3)) and place them above each other as shown below.

$$\begin{array}{ccccccccc} \pi = & (3 & 1 & 2) & (7 & 8 & 4 & 5 & 6) \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \tau = & (5 & 7 & 2) & (6 & 3 & 4 & 8 & 1) \end{array}$$

Shorter cycles should precede longer ones.

2. Read off the rearrangement as

$$\begin{pmatrix} 3 & 1 & 2 & 7 & 8 & 4 & 5 & 6 \\ 5 & 7 & 2 & 6 & 3 & 4 & 8 & 1 \end{pmatrix}$$

or, written in the usual way, as

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 7 & 2 & 5 & 4 & 8 & 1 & 6 & 3 \end{pmatrix} \quad (1.6)$$

3. Produce the most general patterned symbolic matrix for π by

```
?- var_matrix(8,_M), list_permute([2,3,1,5,6,7,8,4],_M,_P),
   transpose(_P,_M), write_matrix(_M).
[_G868, _G871, _G868, _G877, _G877, _G877, _G877, _G877]
[_G868, _G868, _G871, _G877, _G877, _G877, _G877, _G877]
[_G871, _G868, _G868, _G877, _G877, _G877, _G877, _G877]
[_G877, _G877, _G877, _G958, _G961, _G964, _G961, _G958]
[_G877, _G877, _G877, _G958, _G958, _G961, _G964, _G961]
[_G877, _G877, _G877, _G961, _G958, _G958, _G961, _G964]
[_G877, _G877, _G877, _G964, _G961, _G958, _G958, _G961]
[_G877, _G877, _G877, _G961, _G964, _G961, _G958, _G958]
```

¹¹In this notation for types (see [3]), the superscripts stand for the number of times cycles of a particular length occur. The square brackets have nothing to do with Prolog's list notation.

Rename the variables as necessary to see that the above is $\mathbf{M}_1$ (p. 19).

4. Rearrange the *columns* of $\mathbf{M}_1$ according to (1.6) to get

$$\begin{bmatrix} X_6 & X_1 & X_6 & X_6 & X_3 & X_6 & X_3 & X_6 \\ X_6 & X_3 & X_6 & X_6 & X_1 & X_6 & X_3 & X_6 \\ X_6 & X_3 & X_6 & X_6 & X_3 & X_6 & X_1 & X_6 \\ X_2 & X_6 & X_4 & X_4 & X_6 & X_5 & X_6 & X_5 \\ X_5 & X_6 & X_5 & X_4 & X_6 & X_2 & X_6 & X_4 \\ X_4 & X_6 & X_2 & X_5 & X_6 & X_5 & X_6 & X_4 \\ X_4 & X_6 & X_5 & X_2 & X_6 & X_4 & X_6 & X_5 \\ X_5 & X_6 & X_4 & X_5 & X_6 & X_4 & X_6 & X_2 \end{bmatrix}$$

5. Now, using (1.6) again, rearrange the *rows* of the matrix from the previous step.

$$\begin{bmatrix} X_4 & X_6 & X_2 & X_5 & X_6 & X_5 & X_6 & X_4 \\ X_6 & X_3 & X_6 & X_6 & X_1 & X_6 & X_3 & X_6 \\ X_5 & X_6 & X_4 & X_5 & X_6 & X_4 & X_6 & X_2 \\ X_2 & X_6 & X_4 & X_4 & X_6 & X_5 & X_6 & X_5 \\ X_6 & X_3 & X_6 & X_6 & X_3 & X_6 & X_1 & X_6 \\ X_4 & X_6 & X_5 & X_2 & X_6 & X_4 & X_6 & X_5 \\ X_6 & X_1 & X_6 & X_6 & X_3 & X_6 & X_3 & X_6 \\ X_5 & X_6 & X_5 & X_4 & X_6 & X_2 & X_6 & X_4 \end{bmatrix}$$

This is the most general patterned symbolic matrix for τ as is confirmed by the query below.

```
?- var_matrix(8,_M), list_permute([6,5,4,8,7,3,2,1],_M,_P),
   transpose(_P,_M), write_matrix(_M).
[_G868, _G871, _G874, _G877, _G871, _G877, _G871, _G868]
[_G871, _G898, _G871, _G871, _G907, _G871, _G898, _G871]
[_G877, _G871, _G868, _G877, _G871, _G868, _G871, _G874]
[_G874, _G871, _G868, _G868, _G871, _G877, _G871, _G877]
[_G871, _G898, _G871, _G871, _G898, _G871, _G907, _G871]
[_G868, _G871, _G877, _G874, _G871, _G868, _G871, _G877]
[_G871, _G907, _G871, _G871, _G898, _G871, _G898, _G871]
[_G877, _G871, _G877, _G868, _G871, _G874, _G871, _G868]
```

Row-to-row and column-to-column rearrangements obviously retain the total of a numerical matrix. Therefore, most general patterned symbolic matrices belonging to permutations of the same type will evaluate to the same maximum total. This confirms the last of the three results announced in Sect. 1.5.1. The other two are more straightforward. Admissibility (i.e. not having any 1-cycle) is clearly a type-property. Finally, a matrix with distinct rows will be transformed to a such by a row-to-row or column-to-column rearrangement. Therefore, row-distinctness is also a type-property.

1.5.3 Generating Representative Permutations

Generating Permutation Types

The following algorithm, which is from [3, p. 440], is for obtaining all partitions of a number. It will serve as a basis for generating all permutation types for a given problem size. (As mentioned earlier, there is a one-to-one correspondence between partitions of a number and permutation types.)

The following rule is the basis for a method of listing all partitions of n in lexicographic order.¹² The first partition is $[n]$. Suppose the current partition λ has parts $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_r$. Then the next partition is found as follows:

- (i) if $\lambda_r \neq 1$, then the parts of the next partition are $\lambda_1, \lambda_2, \dots, \lambda_{r-1}, \lambda_r - 1, 1$;

¹²The following is an appropriate ordering. For two partitions of n , $p = [1^{\alpha_1} 2^{\alpha_2} \dots n^{\alpha_n}]$ and $r = [1^{\beta_1} 2^{\beta_2} \dots n^{\beta_n}]$, we say that p comes before r (denoted by $p \prec_n r$) if for some $k \in \{1, \dots, n\}$, $\alpha_k > \beta_k$ and $\alpha_i = \beta_i$ for all $i \in \{k+1, \dots, n\}$. For example, $[1^1 3^1 4^1] \prec_8 [1^4 4^1]$ since, more explicitly, $[1^1 2^0 3^1 \text{ } 4^1 5^0 6^0 7^0 8^0] \prec_8 [1^4 2^0 3^0 \text{ } 4^1 5^0 6^0 7^0 8^0]$. (Longest possible identical tail sections are shaded.) In the ascending chain of successors produced by the algorithm, every partition of n appears since $\prec_n$ is a total ordering on the partitions of n .

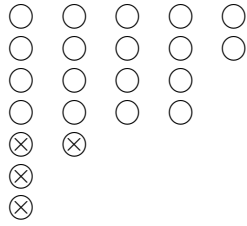
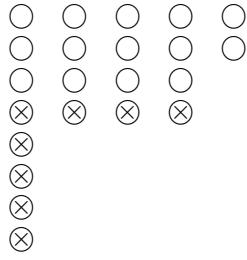
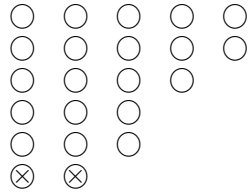
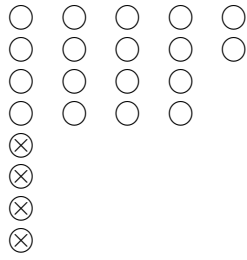
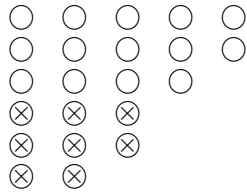
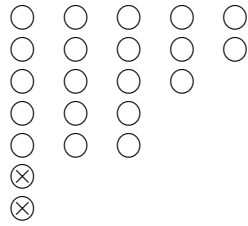
Current Partition	(1)  $[1^2 2^1 4^2 5^2]$	(3)  $[1^4 4^2 5^2]$	(5)  $[2^1 3^2 4^1 5^2]$
Next Partition	(2)  $[1^4 4^2 5^2]$	(4)  $[2^1 3^2 4^1 5^2]$	(6)  $[1^2 3^2 4^1 5^2]$
Step Used	(ii)	(ii)	(i)

Table 1.2: A Ferrers Diagram

- (ii) if $\lambda_r = \lambda_{r-1} = \dots = \lambda_{r-s+1} = 1$ but $\lambda_{r-s} = x \neq 1$, then the parts of the next partition are obtained by replacing $\lambda_{r-s}, \lambda_{r-s+1}, \dots, \lambda_r$ by $x-1, x-1, x-1, \dots, x-1, y$, where $1 \leq y \leq x-1$ and the number of parts $x-1$ is chosen so that the result is a partition of n .

To make the recursive step of this algorithm more accessible, we show in Table 1.2 some typical instances for generating partitions of $n = 22$. *Ferrers Diagrams* ([3]) are used in Table 1.2 to illustrate partitions. Tokens involved in the recursive step are marked ($\times$).

We paraphrase the algorithm in plain English as it may look rather cryptic at first sight. We lay out n tokens to represent the current partition as a Ferrers diagram. The initial pattern will be just a single row of n tokens, denoting the partition $[n]$. All subsequent diagrams will have several rows and (as a rule) longer rows are placed above shorter ones. To decide which of the recursive steps (i) or (ii) applies, we inspect the bottom row. If it contains more than one token, we then remove its last (i.e. rightmost) token and start a new row by placing it below what was hitherto the bottom row. This completes step (i). On the other hand, if the bottom row consists of a single token, we then scan the diagram from bottom to top. There are now two possibilities. We may find that all rows are single-token rows in which case we have found the last partition, $[1^n]$, and stop.

(This has been omitted in the algorithm.) The other possibility is that there is a row containing more than one token. In this case, we remove from the diagram all single-token rows as well as the bottom non-single-token row which has $x (\geq 2)$ tokens, say. (These tokens have been marked in Table 1.2, parts (1) and (3).) The tokens thus removed are now used to build up as many new rows of length $x - 1$ as possible; we place them below the other (undisturbed) tokens. All the remaining tokens, less than $x - 1$, if any, are placed below all the other tokens. This completes step (ii).

Partitions will be represented in our Prolog implementation by lists of pairs; for example, $[(2,1), (3,2), (4,1), (5,2)]$ stands for $[2^1 3^2 4^1 5^2]$.

As a first step towards implementing a type *generator*, we define *next_partition(+Current, -Next)* which for a *Current* partition returns the *Next* partition; for example,

```
?- next_partition([(2,1), (3,2), (4,1), (5,2)], Next).
Next = [ (1, 2), (3, 2), (4, 1), (5, 2)]
```

In (P-1.2) we define those three clauses of *next_partition/2* which are typified by the cases in Table 1.2; the definition of the remaining clauses is asked for in Exercise 1.10.

<i>Current Partition</i>	$[2^3 4^1 6^2]$	...	$[4^3 5^2]$	...
<i>Next Partition</i>	...	$[1^1 3^1 6^3]$	...	$[1^5 2^3 3^1 4^2]$
<i>Step Used</i>	...	...	...	...

<i>Current Partition</i>	...	$[1^3 5^1 7^2]$	...
<i>Next Partition</i>	$[2^1 4^2 6^2]$	...	$[3^3 4^2 5^1]$
<i>Step Used</i>	...	...	...

Table 1.3: Suggested Examples for Exercise 1.10

Prolog Code P-1.2: Three clauses of the predicate *next_partition/2*

```

1 next_partition([(1,Alpha),(2,1)|T],          % Cases (1)-(2) in Table 1.2
2           [(1,NewAlpha)|T]) :-              %
3     NewAlpha is Alpha + 2.                  %

4 next_partition([(1,Alpha1),(L,AlphaL)|T], % Cases (3)-(4) in Table 1.2
5           [(Rest,1),                        %
6           (NewL,Ratio),                     %
7           (L,NewAlphaL)|T]) :-              %
8     L > 2,                                  %
9     AlphaL > 1,                             %
10    NewL is L - 1,                          %
11    Rest is (Alpha1 + L) mod NewL,           %
12    Rest > 0,                               %
13    Ratio is (Alpha1 + L) // NewL,          %
14    NewAlphaL is AlphaL - 1.                %

15 next_partition([(2,1)|T],[(1,2)|T]).      % Cases (5)-(6) in Table 1.2

```

Exercise 1.10. The complete definition of *next_partition/2* comprises ten clauses three of which have been defined already. Typical examples covered by each of the remaining seven clauses are partially shown in Table 1.3. Complete Table 1.3 and then define the missing clauses of *next_partition/2*. (It may be helpful to devise the corresponding Ferrers diagrams by using coins.) ■

The predicate *next_partition/2* returns for a given partition its *successor*. We want, however, a *generator* (also called *enumerator*) of partitions, i.e. a predicate which on backtracking will eventually return *all* partitions. The more general question is as follows: How do we ‘convert’ a successor predicate into a generator? The key to answering this question is by recognizing that this type of problem has been met before. In Exercise 4.6, [9, p. 134], the following definition of *int(+N, ?NextN)* was considered,

```

int(I, I).
int(Last, I) :- succ(Last, New), int(New, I).

```

This definition can be used as a *template* for defining another generator: replace *succ* and *int* respectively by *next_partition* and *part* thus giving,

```
part(P, P).
part(Last, Next) :- next_partition(Last, New), part(New, Next).
```

This will result in an acceptable solution,

```
?- part([(1,2),(2,1),(4,2),(5,2)], P).
P = [ (1, 2), (2, 1), (4, 2), (5, 2)] ;
P = [ (1, 4), (4, 2), (5, 2)] ;
P = [ (2, 1), (3, 2), (4, 1), (5, 2)] ;
...
```

A better idea still is to write a *higher order* predicate, *generator/3*, say, to accomplish the same task for *any* successor predicate. We then have, for example,

```
?- generator(next_partition, [(1,2),(2,1),(4,2),(5,2)], P).
P = [ (1, 2), (2, 1), (4, 2), (5, 2)] ;
P = [ (1, 4), (4, 2), (5, 2)] ;
P = [ (2, 1), (3, 2), (4, 1), (5, 2)] ;
...
```

and

```
?- generator(succ, 7, I).
I = 7 ;
I = 8 ;
I = 9 ;
...
```

We define *generator(+Pred, +Init, ?Element)* in (P-1.3) by

Prolog Code P-1.3: Definition of *generator/3*

```
1 generator(Pred, From, Element) :-
2   retractall(temp(_, _)),
3   assert(temp(First, First)),
4   assert(temp(Last, E) :- (call(Pred, Last, New), temp(New, E))),
5   temp(From, Element).
```

(P-1.3) shows that

- The temporary generator to be defined in the database is named *temp/2*. Possible earlier definitions are removed first.
- Following our template, two clauses of *temp/2* are written to the database. For instance, after running the above example, the database may be inspected thus

```
?- listing(temp).
temp(A, A).
temp(A, B) :- call(succ, A, C), temp(C, B).
```

As the predicate name is open at this stage, *call/3* is used to invoke the predicate in *Pred*. (See inset.)

- Finally, *temp/2*, just written to the database, is invoked and backtracking is used to produce the sequence.

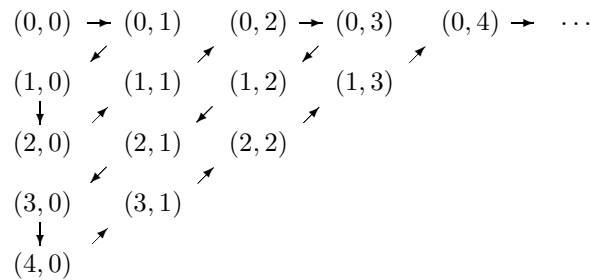


Figure 1.6: Enumeration Scheme for $\{(m, n) : m, n = 0, 1, 2, \dots\}$. (See Exercise 1.12.)

Built-in Predicate: *call/n*, $n = 1, 2, 3, \dots$

call(+Goal) invokes *Goal*. Combine *call/1* with *=../2*, the built in predicate *univ* ([9] or [33]), if the arity of the predicate in *Goal* is known at run time only. Example:

```
?- _Functor = append, _Args = [[1,2],[3],L],
   Goal =.. [_Functor/_Args], call(Goal).
L = [1, 2, 3]
Goal = append([1, 2], [3], [1, 2, 3])
```

Use *call(+Predicate, +Arg1, +Arg2, ...)* to invoke a *Predicate* whose arity is known at compile time. Examples:

```
?- Pred = append, call(Pred,[1,2],[3],L).
Pred = append
L = [1, 2, 3]
?- Pred = append([1,2]), call(Pred,[3],L).
Pred = append([1, 2])
L = [1, 2, 3]
```

call/n is a *higher order* predicate.

Exercise 1.11. Define a predicate *next_int(+Upper, +I, -NextI)* for unifying *NextI* with the value of *I* incremented by 1. The predicate should fail if *Upper* does not exceed *I*. Use *next_int/3* in conjunction with *generator/3* to generate all integers between 3 and 9. ■

Exercise 1.12. Fig. 1.6 indicates an enumeration scheme for all pairs of non-negative integers (the *Cartesian* product). Define *next_pair/2* for returning the *successor* of any given pair. Then use *next_pair/2* in conjunction with *generator/3* for defining an enumerator for the said Cartesian product. ■

Exercise 1.13. (*An improved generator*) The predicate *pairs/1*, defined by

```
pairs((I,J)) :- int(0,Sum), between(0,Sum,I), J is Sum - I.
```

enumerates the pairs of non-negative integers as shown in Fig. 1.7.¹³ It will return on backtracking all pairs starting from (0, 0).

```
?- pairs(P).
P = 0, 0 ;
P = 0, 1 ;
P = 1, 0 ;
P = 0, 2 ;
P = 1, 1 ;
...
```

An *alternative* implementation of *pairs/1* may conceivably be obtained by replacing in its definition the predicates *int/2* and *between/3* by their respective definitions using *generator/3*:

```
pairs_alt((I,J)) :- generator(succ,0,Sum),
                      generator(next_int(Sum),0,I),
                      J is Sum - I.
```

Testing will reveal, however, that this implementation is flawed. The problem is due to the use by *generator/3* of the same name *temp* for predicates written to the database.

¹³The built-in predicate *between/3* is described in [9, p. 41].

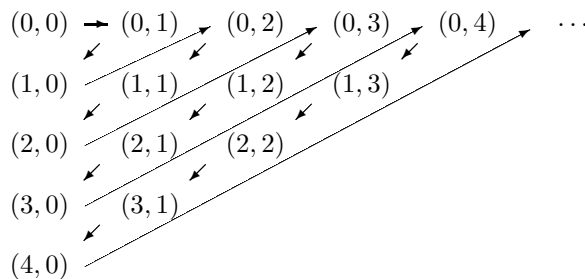


Figure 1.7: Enumeration Scheme for $\{(m, n) : m, n = 0, 1, 2, \dots\}$. (See Exercise 1.13.)

(The call to `generator(next_int(Sum), 0, I)` will interfere with that of `generator(succ, 0, Sum)`.) The problem could be avoided though if `generator/3` created temporary predicates with a *different* and *unique* name every time it is invoked.

Define such an improved version of `generator/3`.

Hint. It is suggested that the temporary predicates be named `temp_0`, `temp_1`, etc. You should use the built-in predicate `current_predicate/2` (described in the SWI manual [33]) for finding out whether a proposed new predicate name is available. Use `concat_atom/2` [9, p. 126] for constructing new predicate names. ■

Admissible Representative Permutations

How many permutation types will have to be considered for the original 8×8 problem? This is easily found out by a query,

```
?- bagof(_P, generator(next_partition, [(8, 1)], _P), _Ps),
   length(_Ps, NTypes).
NTypes = 22
```

The number 22 is further reduced by concentrating on *admissible* permutations, i.e. on those without a 1-cycle; the types of these we obtain by¹⁴

```
?- bagof(_P, _I^_A^_T^(generator(next_partition, [(8, 1)], _P),
   _P = [(_I, _A) | _T], _I > 1), _Ps).
Ps = [[(8, 1)], [(2, 1), (6, 1)], [(3, 1), (5, 1)], [(4, 2)],
      [(2, 2), (4, 1)], [(2, 1), (3, 2)], [(2, 4)]]
```

We therefore have to consider here a mere 7 types. (Contrast this with the 14,833 admissible permutations considered earlier!) All we have to do now is to create for each admissible type a representative permutation.

Suppose we want to construct a representative permutation for the type $[2^1 3^3 5^1]$, a partition of 16. An example permutation of this type in the cycle notation is obtained by simply grouping the elements of $\{1, \dots, 16\}$ according to the length of the cycles needed:

$$(1\ 2)(3\ 4\ 5)(6\ 7\ 8)(9\ 10\ 11)(12\ 13\ 14\ 15\ 16) \quad (1.7)$$

¹⁴This query gives rise to `ad_partition(+N, ?P)`, a predicate for generating (and testing) admissible partitions of N :

```
ad_partition(N, [(I, A) | T]) :- generator(next_partition, [(N, 1)], [(I, A) | T]),
                               I > 1.
```

Using the two-line notation, we rewrite this as

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 2 & 1 & 4 & 5 & 3 & 7 & 8 & 6 & 10 & 11 & 9 & 13 & 14 & 15 & 16 & 12 \end{pmatrix} \quad (1.8)$$

which then in the Prolog implementation will be denoted by

$$[2, 1, 4, 5, 3, 7, 8, 6, 10, 11, 9, 13, 14, 15, 16, 12] \quad (1.9)$$

The Prolog implementation of (1.7)–(1.9) is in three steps:

- (a) A predicate *split*(+N,+Type,-S) is used for partitioning $[1, \dots, 16]$ into a list of sublists *S* according to *Type*:¹⁵

```
?- split(16, [(2,1), (3,3), (5,1)], _S), write_term(_S, []).
[[1,2], [3,4,5], [6,7,8], [9,10,11], [12,13,14,15,16]]
```

split/3 is defined below in terms of an auxiliary predicate *split*/4 which itself uses the accumulator technique.

```
split(N,Type,S) :- from_to(1,N,L), split(L,Type,[],S).
```

Exercise 1.14. Define *split*/4. (Some suggested hand computations are shown in Fig. 1.8, p. 45.) ■

- (b) *maplist*/3 is applied to rotate each sublist in the above list-of-lists.¹⁶

```
?- split(16, [(2,1), (3,3), (5,1)], _S), maplist(rotate, _S, _R),
   write_term(_R, []).
[[2,1], [4,5,3], [7,8,6], [10,11,9], [13,14,15,16,12]]
```

- (c) Finally, *flatten*/2 is used to obtain the list in (1.9).

(a)–(c) give rise to *rep_perm*(+N,+Type,-Perm), a predicate for finding a representative permutation of a given type.

```
rep_perm(N,Type,Perm) :- split(N,Type,S),
                        maplist(rotate,S,R),
                        flatten(R,Perm).
```

1.5.4 Finishing Touches

Based on the ideas in Sect. 1.5.3, we are now in a position to define a new version of the predicate *square*/5, defined in (P-1.1); the new definition is shown in (P-1.4). Now the queries from Sect. 1.4.10 may be completed as before and with a much reduced computing time. For example, for a 14×14 board we find by a near instantaneous response that the maximum total is 4900. (The earlier version won't solve this problem due to memory shortage and excessive computing time.)

¹⁵The first argument of *split*/3 is redundant as it can be computed from *Type*. Not having to recompute it, however, will save computing time.

¹⁶Prolog implementations of list rotation are discussed in [5], [8] and [9].

Prolog Code P-1.4: Definition of square2/5

```
1 square2(Size,M,Total,Freq,Perm) :- var_matrix(Size,M),  
2                                   ad_partition(Size,Type),  
3                                   rep_perm(Size,Type,Perm),  
4                                   list_permute(Perm,M,P),  
5                                   transpose(P,M),  
6                                   distinct(M),  
7                                   eval_matrix(M,Freq),  
8                                   total(Freq,Total).
```

```

split([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16], [(2,1),(3,3),(5,1)], [], S) ~>
split([3,4,5,6,7,8,9,10,11,12,13,14,15,16], [(2,0),(3,3),(5,1)], [[1,2]], S) ~>
split([3,4,5,6,7,8,9,10,11,12,13,14,15,16], [(3,3),(5,1)], [[1,2]], S) ~>
split([6,7,8,9,10,11,12,13,14,15,16], [(3,2),(5,1)], [[3,4,5], [1,2]], S) ~>
split([9,10,11,12,13,14,15,16], [(3,1),(5,1)], [[6,7,8], [3,4,5], [1,2]], S) ~>
split([12,13,14,15,16], [(3,0),(5,1)], [[9,10,11], [6,7,8], [3,4,5], [1,2]], S) ~>
split([12,13,14,15,16], [(5,1)], [[9,10,11], [6,7,8], [3,4,5], [1,2]], S) ~>
split([], [(5,0)], [[12,13,14,15,16], [9,10,11], [6,7,8], [3,4,5], [1,2]], S) ~>
S = [[1,2], [3,4,5], [6,7,8], [9,10,11], [12,13,14,15,16]] ~> success

```

Figure 1.8: Suggested Hand Computations for *split/4*

Chapter 2

Blind Search

Many problems in Artificial Intelligence (AI) can be formulated as network search problems. The crudest algorithms for solving problems of this kind, the so called *blind search algorithms*, use the network's connectivity information only. We are going to consider examples, applications and Prolog implementations of blind search algorithms in this chapter.

Since implementing solutions of problems based on search usually involves code of some complexity, *modularization* will enhance clarity, code reusability and readability. In preparation for these more complex tasks in this chapter, Prolog's module system will be discussed in the next section.

2.1 Digression on the Module System in Prolog

In some (mostly larger) applications there will be a need to use *several* input files for a Prolog project. We have met an example thereof already in Fig. 3.5 of [9, p. 85] where *consult/1* was used as a *directive* to include in the database definitions of predicates from other than the top level source file. As a result, all predicates thus defined became *visible* to the user: had we wished to introduce some further predicates, we would have had to choose the names so as to avoid those already used. Clearly, there are situations where it is preferable to make available (that is, to *export*) only those predicates to the outside world which will be used by other non-local predicates and to hide the rest. This can be achieved by the built-in predicates *module/2* and *use_module/1*.

As an illustrative example, consider the network in Fig. 2.1.¹ The network connectivity in `links.pl` is defined by the predicate *link/2* which uses the auxiliary predicate *connect/2* (Fig. 2.2).

The first line of `links.pl` is the *module directive* indicating that the *module name* is *edges* and that the predicate *link/2* is to be *exported*. All other predicates defined in `links.pl` (here: *connect/2*) are *local* to the module and (normally) not visible outside this module.

Suppose now that in *some other* source file, *link/2* is used in the definition of some new predicate (Fig. 2.3). Then, the (visible) predicates from `links.pl` will be *imported* by means of the directive

```
:- use_module(links).2
```

The new predicate thus defined may be used as usual:

¹This is a network from the AI-classic [34].

²Notice that the argument in *use_module/1* is the filename without the `.pl` extension.

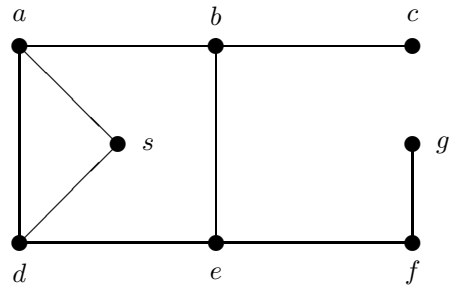


Figure 2.1: A Network

```
?- consult(df1).  
% links compiled into edges 0.00 sec, 1,644 bytes  
% df1 compiled 0.00 sec, 3,208 bytes  
Yes  
?- successors(a,L).
```

```

:- module(edges,[link/2]).

connect(a,b).  connect(a,d).  connect(a,s).
connect(b,c).  connect(b,e).
connect(d,e).  connect(d,s).
connect(e,f).
connect(f,g).

link(Node1,Node2) :- connect(Node1,Node2).
link(Node1,Node2) :- connect(Node2,Node1).

```

Figure 2.2: The File `links.pl`

```

:- use_module(links).
...
...
successors(Node,SuccNodes) :-
    findall(Successor,link(Node,Successor),SuccNodes).

```

Figure 2.3: Fragment of the File `df1.pl`

```

L = [b, d, s] ;
No

```

In our example, the predicate `connect/2` will not be available for use (since it is local to the module `edges` that resides in `links.pl`). A local predicate may be accessed, however, by *prefixing* its name by the module name in the following fashion:³

```

?- edges:connect(a,N).
N = b ;
N = d ;
N = s ;
No

```

(Notice the distinct uses of the module name and the name of the file in which the module resides.)

2.2 Basic Search Problem

Let us assume that for the network in Fig. 2.1 we want to find a *path* from the *start node* *s* to the *goal node* *g*. The search may be conducted by using the (associated) *search tree* shown in Fig. 2.4. It is seen that the

³SWI-Prolog will suggest a correction if the predicate name is used without the requisite prefix:

```

?- connect(a,N).
Correct to: edges:connect(a, N)? yes
N = b ;
...

```

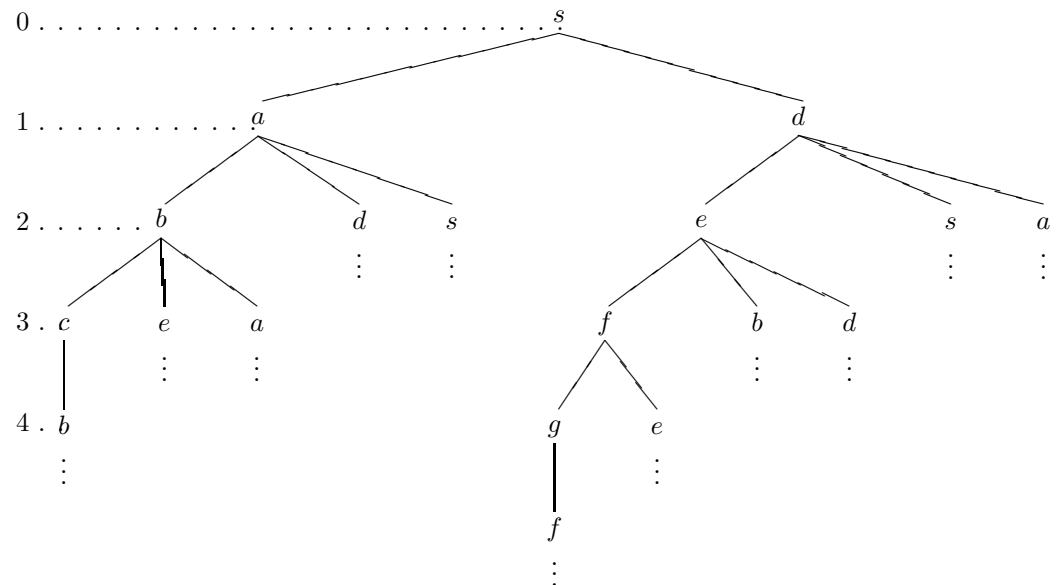


Figure 2.4: The Search Tree

search tree is infinite but highly repetitive. The start node s is at the *root node* (level 0). At level 1, all tree nodes are labelled by those network nodes which can be reached in one step from the start node. In general, a node labelled n in the tree at level ℓ has *successor* (or *child*) nodes labelled $s_1, s_2, \dots$ if the nodes $s_1, s_2, \dots$ in the network can be reached in one step from node n . These successor nodes are said to be at level $\ell + 1$. The node labelled n is said to be a *parent* of the nodes $s_1, s_2, \dots$. In Fig. 2.4, to avoid repetition, those parts of the tree which can be generated by expanding a node from some level above have been omitted.

Some Further Terminology

- The connections between the nodes in a network are called *links*.
- The connections in a tree are called *branches*.
- In a tree, a node is said to be the *ancestor* of another if there is a chain of branches (upwards) which connects the latter node to the former. In a tree, a node is said to be a *descendant* of another node if the latter is an ancestor of the former.

In Fig. 2.5 we show, for later reference, the fully developed (and 'pruned') search tree. It is obtained from Fig. 2.4 by arranging that in any chain of branches (corresponding to a path in the network) there should be no two nodes with the same label (implying that in the network no node be visited more than once). All information pertinent to the present problem is recorded thus in the file `links.pl` (Fig. 2.2) by `link/2`. Notice that the order in which child nodes are generated by `link/2` will govern the development of the trees in Figs. 2.4 and 2.5: children of the same node are written down from left to right in the order as they would be obtained by backtracking; for example, the node labelled d at level 1 in Fig. 2.4 is expanded by

```
?- link(d,Child).
```

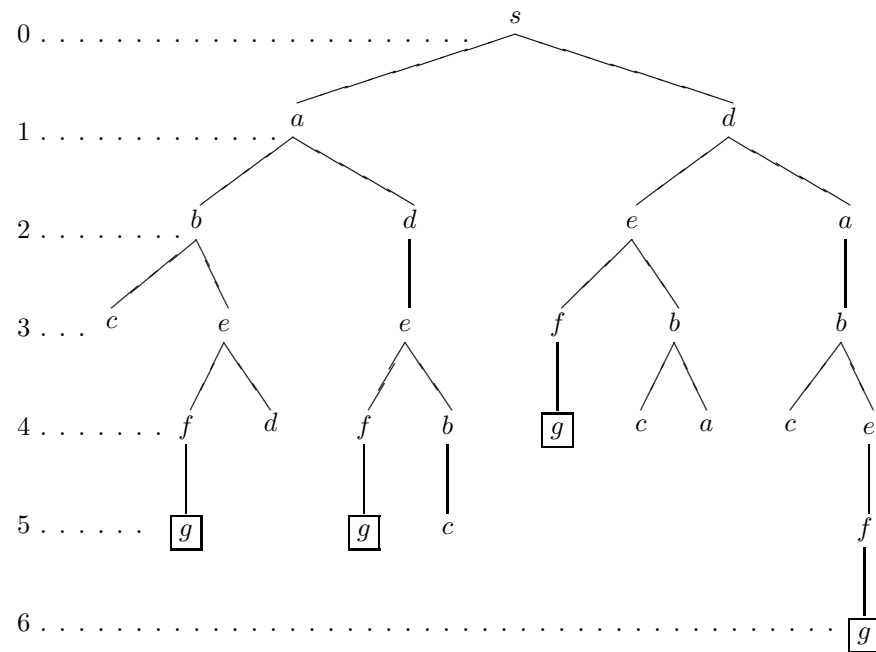


Figure 2.5: The Pruned Search Tree

```
Child = e ;  
Child = s ;  
Child = a ;  
No
```

(The same may be deduced, of course, by inspection from `links.pl`, Fig. 2.2.) *link/2* will serve as input to the implementations of the search algorithms to be discussed next.

2.3 Depth First Search

The most concise and easy to remember illustration of Depth First is by the *conduit model* (Fig. 2.6). We start with the search tree in Fig. 2.5 which is assumed to be a network of pipes with inlet at the root node *s*. The tree is rotated by 90° counterclockwise and connected to a valve which is initially closed. The valve is then opened and the system is observed as it gets flooded under the influence of gravity. The order in which the nodes are wetted corresponds to Depth First.

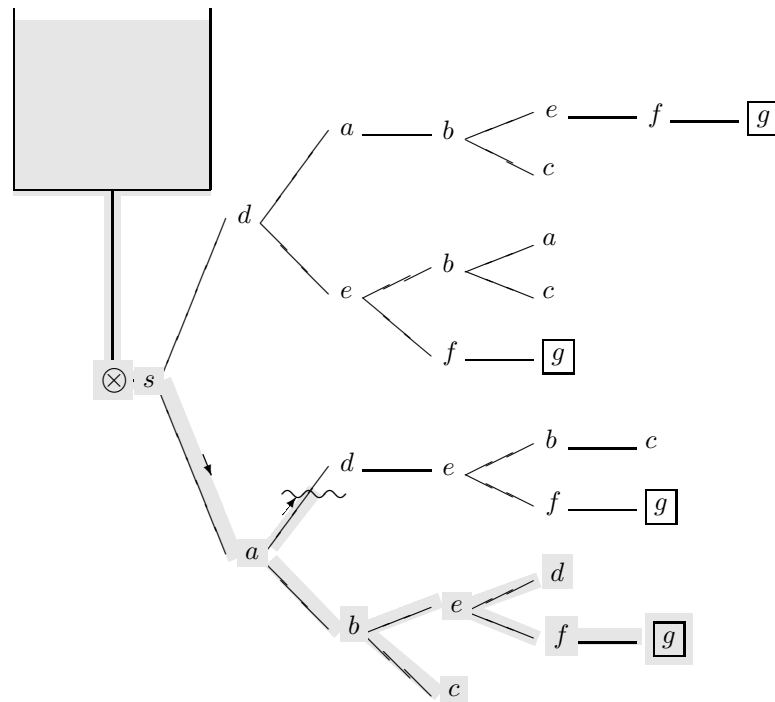


Figure 2.6: Depth First Search – The Conduit Model

```

:- use_module(links).

path(Node1,Node2,[Node1,Node2]) :- link(Node1,Node2).
path(Node1,Node2,[Node1|Path32]) :- link(Node1,Node3),
    write('visiting node '), write(Node3), nl,
    path(Node3,Node2,Path32).

```

Figure 2.7: The File `naive.pl`

2.3.1 Naïve Solution

We may be tempted to use Prolog's backtracking mechanism to furnish a solution by recursion; our attempt is shown in Fig. 2.7.⁴ However, it turns out that the implementation does not work due to cycling in the network. The query shown below illustrates the problems arising.

```

?- path(s,g,Path).
visiting node a
visiting node b
visiting node c
visiting node b
visiting node c
...
Action (h for help) ? abort
% Execution Aborted

```

2.3.2 Incremental Development Using an Agenda

We implement Depth First search incrementally using a new approach. The idea is keeping track of the nodes to be visited by means of a list, the so called *list of open nodes*, also called the *agenda*. This book-keeping measure will turn out to be amenable to generalization; in fact, it will be seen that the various search algorithms differ only in the way the agenda is updated.

First Version

A first, preliminary, form of Depth First search is stated in Algorithm 2.3.1. The definition of the corresponding predicate, *depth_first/2*, is shown in Fig. 2.8. (At this stage, we are attempting an implementation which merely succeeds once the goal node is found.)

⁴The shaded entries facilitate explanatory screen displays only.

Algorithm 2.3.1: DEPTHFIRST(*StartNode*, *GoalNode*)**comment:** First tentative implementation of Depth First Search*RootNode* $\leftarrow$ *StartNode**OpenList* $\leftarrow$ [*RootNode*]*[H|T]* $\leftarrow$ *OpenList***while** *H* $\neq$ *GoalNode*

	$\left\{ \begin{array}{l} \textit{SuccList} \leftarrow \text{successors of } H \\ \textit{OpenList} \leftarrow \textit{SuccList} \mathbin{++} T \end{array} \right.$
do	$\left\{ \begin{array}{l} \text{if } \textit{OpenList} = [] \\ \quad \text{then return (failure)} \\ \textit{[H T]} \leftarrow \textit{OpenList} \end{array} \right.$

return (*success*)

What is the crucial feature of this algorithm? It is the way the list of open nodes is manipulated. There are two possibilities:

```

:- use_module(links).

depth_first(Start,Goal) :- dfs_loop([Start],Goal).

dfs_loop([Goal|_],Goal).
dfs_loop([CurrNode|OtherNodes],Goal) :-
    successors(CurrNode,SuccNodes),
    write('Node '), write(CurrNode),
    write(' is being expanded. '),
    append(SuccNodes,OtherNodes,NewOpenNodes),
    write('Successor nodes: '), write(SuccNodes), nl,
    write('Open nodes: '), write(NewOpenNodes), nl,
    dfs_loop(NewOpenNodes,Goal).

successors(Node,SuccNodes) :-
    findall(Successor,link(Node,Successor),SuccNodes).

```

Figure 2.8: The File df1.pl

- *Inspection.* We may inspect the agenda's head to see whether it is the goal node.
- *Updating.* If the head is not the goal node, we determine the head's successor or successors. They are collected into a list, *SuccList*, say, (which may well be empty) and a new agenda will be formed by appending the tail of the old agenda to *SuccList*. The *order* of entries in the list just created is essential: the successors of the most recently visited node are placed *to the front*, thereby becoming candidates for more immediate attention.

As mentioned earlier, search algorithms differ from each other only in the way the list of open nodes is updated. The updating mechanism of Depth First is on a last-in-first-out (LIFO) basis.

The (unsatisfactory) behaviour of *depth_first/2* in the present form is exemplified in Fig. 2.9. Obviously, the order of the nodes' expansion is as expected but we descend into ever greater depths of (the leftmost part of) the tree in Fig. 2.4. There are two possible solutions to this problem – they will be discussed below.

Using a List of 'Closed Nodes'

The underlying idea of this approach is that a node on the search tree should not be included in the open list (again) if a node with the same label has ever been visited before. The examples below will show (and indeed a moment of reflection should confirm) that this method may not find all goal nodes (or all paths to the goal node(s)). The realization of the idea is as follows. Once we remove *H* from the list of open nodes (Algorithm 2.3.1) we should include *H* into another list, the list of *closed* nodes, indicating that it should not be expanded (i.e. included in the list of open nodes) ever again. This version of Depth First search is shown as Algorithm 2.3.2. The corresponding Prolog program, *df2.pl*, is shown in Fig. 2.10. Finally, an interactive session with this second version of *depth_first/2* is shown in Fig. 2.11. The missing (shaded) parts in Fig. 2.10 are goals for displaying information on the progress of the search as seen in Fig. 2.11.

Exercise 2.1. Complete the code in Fig. 2.10 such that the response shown in Fig. 2.11 is achieved. ■

```

?- depth_first(s,g).
Node s is being expanded. Successor nodes: [a, d]
Open nodes: [a, d]
Node a is being expanded. Successor nodes: [b, d, s]
Open nodes: [b, d, s, d]
Node b is being expanded. Successor nodes: [c, e, a]
Open nodes: [c, e, a, d, s, d]
...
Action (h for help) ? abort
% Execution Aborted

```

Figure 2.9: Illustrative Query for *depth_first/2* – First Version

Algorithm 2.3.2: DEPTHFIRST(*StartNode*, *GoalNode*)

comment: Depth First Search with a List of Closed Nodes

```

RootNode ← StartNode
OpenList ← [RootNode]
ClosedList ← []
[H|T] ← OpenList
while  $H \neq \text{GoalNode}$ 
    {
        SuccList ← successors of  $H$  (1)
        OpenList ←  $(\text{SuccList} \cap \text{ClosedList}^c) \uplus T$  (2)
        ClosedList ←  $[H | \text{ClosedList}]$  (3)
    }
    do {
        if OpenList = []
        then return (failure)
        [H|T] ← OpenList
    }
return (success)

```

```

:- use_module(links).

depth_first(Start,Goal) :- ..., % clause 0
                           dfs_loop([Start],[],Goal). %

dfs_loop([Goal|_],_,Goal) :- ... . % clause 1

dfs_loop([CurrNode|OtherNodes],ClosedList,Goal) :- % clause 2
    successors(CurrNode,SuccNodes), } ← Implements (1)
    ..., %
    findall(Node,(member(Node,SuccNodes), } ← Implements (2)
        not(member(Node,ClosedList))),Nodes), %
    append(Nodes,OtherNodes,NewOpenNodes), %
    ..., %
    dfs_loop(NewOpenNodes,[CurrNode|ClosedList],Goal). %

successors(Node,SuccNodes) :- ↑ Implements (3)
    findall(Successor,link(Node,Successor),SuccNodes).

```

Figure 2.10: The File df2.pl

```

?- depth_first(s,g).
Open: [s], Closed: []
Node s is being expanded. Successors: [a, d]
Open: [a, d], Closed: [s]
Node a is being expanded. Successors: [b, d, s]
Open: [b, d, d], Closed: [a, s]
Node b is being expanded. Successors: [c, e, a]
Open: [c, e, d, d], Closed: [b, a, s]
Node c is being expanded. Successors: [b]
Open: [e, d, d], Closed: [c, b, a, s]
Node e is being expanded. Successors: [f, b, d]
Open: [f, d, d, d], Closed: [e, c, b, a, s]
Node f is being expanded. Successors: [g, e]
Open: [g, d, d, d], Closed: [f, e, c, b, a, s]
Goal found: g
Yes

```

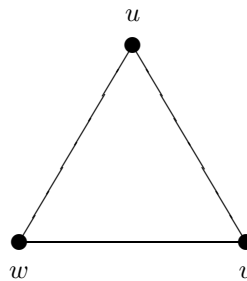
Figure 2.11: Illustrative Query for *depth_first/2* – Second Version

Figure 2.12: The New Network Component

Exercise 2.2. Suppose we want to model a network which arises by augmenting the graph in Fig. 2.1 with the one shown in Fig. 2.12, p. 59. (The new network thus comprises two *unconnected* components.)

- (a) Augment the database in Fig. 2.2 to reflect the connectivity of the new network.
- (b) Write down hand computations for the queries
 - (i) `?- depth_first(d, c).`
 - (ii) `?- depth_first(u, c).`

■

The predicate *depth_first/2* from *df2.pl* (Fig. 2.10) finds a goal node (if there is one) but does not return the corresponding path. (We ignore the shaded clauses in Fig. 2.10 as they are there for explanatory reasons only.) A new, improved version, *depth_first(+Start, +Goal, -Path)*, say, should return also the *Path* found,

given the *Start* node and the *Goal* node. We modify the auxiliary predicate *dfs_loop/3* from *df2.pl* in two ways.

- Now, its first argument will take the list of open *paths* (and not that of open nodes). This is the argument where we accumulate (maintain) the agenda.
- Into an additional (fourth) argument will the path from *Start* to *Goal* be copied as soon as it appears at the head of the agenda. The search is then finished.
- The second and third arguments of *dfs_loop/4* will hold, as before, the list of closed nodes and the goal node, respectively.

The hand computations in Fig. 2.13, p. 61, indicate the required behaviour of the new version of *depth_first/3*.

Paths will be represented by the lists of nodes visited; internally, they will be read from right to left. For example, the list *[g, f, e, b, a, s]* will stand for the path $s \rightarrow a \rightarrow b \rightarrow e \rightarrow f \rightarrow g$. In Fig. 2.13, *all* paths we have been temporarily admitted to the agenda which arise by expanding the *head of the head* of the agenda. (Expanding a node means finding its successors.) Immediately after expansion, however, those paths have been removed (indicated by */////*) whose head features in the list of closed nodes in the line *above*. To implement the corresponding predicate *depth_first/3* (Fig. 2.14, p. 63), Algorithm 2.3.3 has been used with an auxiliary procedure *EXTENDPATH*.

```

depth_first(s, g, Path)  $\rightsquigarrow$ 
dfs_loop([[s]], [], g, Path)  $\rightsquigarrow$ 
dfs_loop([[a,s], [d,s]], [s], g, Path)  $\rightsquigarrow$ 
dfs_loop([[b,a,s], [d,a,s], [s,d,s], [d,s]], [a,s], g, Path)  $\rightsquigarrow$ 
dfs_loop([[c,b,a,s], [e,b,a,s], [d,a,s], [d,s]], [b,a,s], g, Path)  $\rightsquigarrow$ 
dfs_loop([[b,d,b,d], [e,b,a,s], [d,a,s], [d,s]], [c,b,a,s], g, Path)  $\rightsquigarrow$ 
dfs_loop([[f,e,b,a,s], [b,d,b,d], [d,e,b,a,s], [d,a,s], [d,s]], [e,c,b,a,s], g, Path)  $\rightsquigarrow$ 
dfs_loop([[g,f,e,b,a,s], [e,f,d,b,d], [d,e,b,a,s], [d,a,s], [d,s]], [f,e,c,b,a,s], g, Path)  $\rightsquigarrow$ 
dfs_loop([[g,f,e,b,a,s], [d,e,b,a,s], [d,a,s], [d,s]], [f,e,c,b,a,s], g, [g,f,e,b,a,s])  $\rightsquigarrow$ 
depth_first(s, g, [g,f,e,b,a,s])  $\rightsquigarrow$  success

```

Figure 2.13: Hand Computations for the Query $?- \text{depth_first}(s, g, \text{Path})$.

Exercise 2.3. Define *extend_path(+Nodes, +Path, -NewPaths)* from Algorithm 2.3.3. ■

Algorithm 2.3.3: DEPTHFIRST(*StartNode*, *GoalNode*)

comment: Depth First with Closed Nodes and Open Paths

procedure EXTENDPATH($[x_1, \dots, x_N]$, *list*)

comment: To return [] if the first argument is []

for $i \leftarrow 1$ **to** N

do $\{list_i \leftarrow [x_i|list]$

return $([list_1, \dots, list_N])$

main

$RootNode \leftarrow StartNode$

$OpenPaths \leftarrow [[RootNode]]$

$ClosedNodes \leftarrow []$

$[[H|T]|TailOpenPaths] \leftarrow OpenPaths$

while $H \neq GoalNode$

$SuccList \leftarrow$ successors of H

$NewOpenNodes \leftarrow (SuccList \cap ClosedList^c)$

$NewPaths \leftarrow EXTENDPATH(NewOpenNodes, [H|T])$

do $\left\{ \begin{array}{l} OpenPaths \leftarrow NewPaths \uparrow\uparrow TailOpenPaths \\ \text{if } OpenPaths = [] \\ \quad \text{then return (failure)} \\ [[H|T]|TailOpenPaths] \leftarrow OpenPaths \end{array} \right.$

$Path \leftarrow REVERSE([H|T])^5$

output ($Path$)

In the query shown below, the predicate *depth_first/3* thus defined finds the *leftmost* path to the goal node in Fig. 2.4. On backtracking, no further paths to the goal node will be found.

```
?- depth_first(s,g,Path).
Path = [s, a, b, e, f, g] ;
No
```

Path Checking

This technique allows *all* paths to the goal node to be found. We do not use a list of closed nodes here. Instead, upon prefixing the head of the agenda by each of the successors of its head, we check for each of the lists thus created whether it is a path. In Algorithm 2.3.4, p. 64, this test is carried out by the as yet unspecified procedure ISPATH. Usually, paths will be required not to contain cycles. Then, the procedure ISPATH checks for distinct entries of the argument list.⁶

The main body of Algorithm 2.3.4 has been implemented by the predicate *depth_first/4*, defined in df4.pl, Fig. 2.15, p. 65. A few noteworthy features of this implementation of Depth First are as follows.

⁵For a pseudocode of REVERSE, see [9, p. 24].

⁶By induction, this test simplifies to showing that the head of a putative path is not an entry in its tail.

```

:- use_module(links).

depth_first(Start,Goal,PathFound) :-
    dfs_loop([[Start]],[],Goal,PathFoundRev),
    reverse(PathFoundRev,PathFound).

dfs_loop([[Goal|PathTail]|_],_,Goal,[Goal|PathTail]).

dfs_loop([[CurrNode|T]|Others],ClosedList,Goal,PathFound) :-
    successors(CurrNode,SuccNodes),
    findall(Node,(member(Node,SuccNodes),
        not(member(Node,ClosedList))),Nodes),
    extend_path(Nodes,[CurrNode|T],Paths),
    append(Paths,Others,NewOpenPaths),
    dfs_loop(NewOpenPaths,[CurrNode|ClosedList],Goal,PathFound).

successors(Node,SuccNodes) :-
    findall(Successor,link(Node,Successor),SuccNodes).

% auxiliary predicate extend_path/3 ...
...

```

Figure 2.14: The File `df3.pl` – Depth First with Closed Nodes and Open Paths

- The arguments of *depth_first(+Start, +G_Pred, +C_Pred, -PathFound)*, the main predicate, play the following rôle:
 - As before, *Start* is unified with the start node.
 - *G_Pred* is unified with the name of the goal *predicate*. (In earlier implementations, a goal *node* was expected.) Due to this generalization, in more complex applications, now a goal node may be specified by a *condition*. Several goal nodes may thus also be accounted for.
 - The third argument, *C_Pred*, is unified with the name of the connectivity predicate which in earlier implementations was *link/2*. Greater flexibility is afforded by this additional argument. In the example query in Fig. 2.17, p. 66, the connectivity predicate *link/2* is used which is defined in *links.pl* (see p. 49) from where it is imported by the first *use_module/1* directive in *df4.pl*.
 - Finally, on return, the last argument is unified with the path found.

Algorithm 2.3.4: DEPTHFIRST(*StartNode*, *G_Pred*, *C_Pred*)

comment: Depth First with Path Checking.

Procedures are assumed available for

- Testing whether a path is a goal path by using the procedure *in G_Pred*;
- Finding successors of a node by using the connectivity procedure *in C_Pred*.

procedure ISPATH(*list*)

comment: Returns a Boolean value.

Is application specific.

:

main

RootNode ← *StartNode*

OpenPaths ← [[*RootNode*]]

[[*H|T*]|*TailOpenPaths*] ← *OpenPaths*

while [*H|T*] is not a goal path

$$\left\{ \begin{array}{l} \text{SuccList} \leftarrow \text{successors of } H \\ \text{ONodes} \leftarrow \text{list of } S \in \text{SuccList} \text{ with } \text{ISPATH}([S, H|T]) \\ \text{NewPaths} \leftarrow \text{EXTENDPATH}(\text{ONodes}, [H|T]) \\ \text{do } \left\{ \begin{array}{l} \text{OpenPaths} \leftarrow \text{NewPaths} \mathbin{++} \text{TailOpenPaths} \\ \text{if } \text{OpenPaths} = [] \\ \quad \text{then return (failure)} \\ \text{[[H|T]|TailOpenPaths]} \leftarrow \text{OpenPaths} \end{array} \right. \end{array} \right.$$

Path ← REVERSE([*H|T*])

output (*Path*)

```

:- use_module(links).
:- use_module(searchinfo).

depth_first(Start,G_Pred,C_Pred,PathFound) :-
    dfs_loop([[Start]],G_Pred,C_Pred,PathFoundRev),
    reverse(PathFoundRev,PathFound).

dfs_loop([Path|_],G_Pred,_,Path) :- call(G_Pred,Path).
dfs_loop([[CurrNode|T]|Others],G_Pred,C_Pred,PathFound) :-
    successors(C_Pred,CurrNode,SuccNodes),
    findall(Node,(member(Node,SuccNodes),
        is_path([Node,CurrNode|T])),Nodes),
    extend_path(Nodes,[CurrNode|T],Paths),
    append(Paths,Others,NewOpenPaths),
    dfs_loop(NewOpenPaths,G_Pred,C_Pred,PathFound).

% auxiliary predicates ...

successors(C_Pred,Node,SuccNodes) :-
    findall(Successor,call(C_Pred,Node,Successor),SuccNodes).

extend_path([],_,[]).
extend_path([Node|Nodes],Path,[[Node|Path]|Extended]) :-
    extend_path(Nodes,Path,Extended).

```

Figure 2.15: The File df4.pl – Depth First with Path Checking

```

:- module(info,[goal_path/1, is_path/1]).

goal_path([g|_]).

is_path([H|T]) :- not(member(H,T)).

```

Figure 2.16: The File searchinfo.pl

```
?- consult(df4).
% links compiled into edges 0.05 sec, 1,900 bytes
% searchinfo compiled into info 0.00 sec, 1,016 bytes
% df4 compiled 0.05 sec, 4,944 bytes
Yes
?- depth_first(s,goal_path,link,Path).
Path = [s, a, b, e, f, g] ;
Path = [s, a, d, e, f, g] ;
Path = [s, d, e, f, g] ;
Path = [s, d, a, b, e, f, g] ;
No
```

Figure 2.17: Interactive Session for *depth_first/4* – Path Checking

- The **while** loop in Algorithm 2.3.4 is implemented by *dfs_loop/4*. It uses the predicate *is_path/1*, an implementation of the procedure ISPATH.

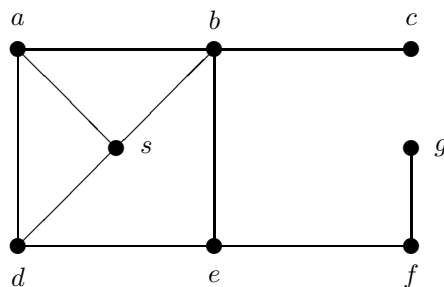


Figure 2.18: A Network (see Exercise 2.4, p. 67)

This predicate is imported from `searchinfo.pl` (Fig. 2.16, p. 65) by the second `use_module/1` directive in `df4.pl`. In the present version of `is_path/1`, paths are defined to be lists with distinct entries.

- `call/2` and `call/3`, are used (see p. 40) to invoke the imported predicates `goal_path/1` and `link/2` at run time.
- It is seen from Fig. 2.17 that on backtracking all paths to the goal node are found.

Exercise 2.4. A new network is shown in Fig. 2.18, p. 67.

- Augment the file `links.pl` to reflect the connectivity of the new network.
- Suppose we want to find all paths from `s` to `g` such that no edge is traversed more than once but we don't mind visiting nodes several times. Define a new version of `is_path/1` in `searchinfo.pl` to this new specification.
- Run `depth_first/4` to find all paths from `s` to `g`.

■

Exercise 2.5. Rewrite the definition of `depth_first/4` in Fig. 2.15 using difference lists.

Hints. You should represent paths, as before, by ordinary lists and write the *agenda* in terms of difference lists. Modify accordingly the predicates `dfs_loop` and `extend_path`. The latter should be invoked by a new version of `depth_first/4`, called `depth_first_dl/4`. You should confirm the advantage of using difference lists by a sample session. (The model solution is found in the file `df.pl` along with the old version based on ordinary lists.)

■

2.4 Breadth First Search

Another blind search algorithm is Breadth First. It visits the nodes of the search tree level by level from left to right as indicated in Fig. 2.19. It always finds a shortest path to the goal node. Now the agenda is updated on a first-in-first-out (FIFO) basis, thus the successors of a node just expanded will be put to the *end* of the list of open nodes.

The definition of `breadth_first/4` in Fig. 2.20, p. 69, is arrived at by minor modifications of the code in Fig. 2.15:

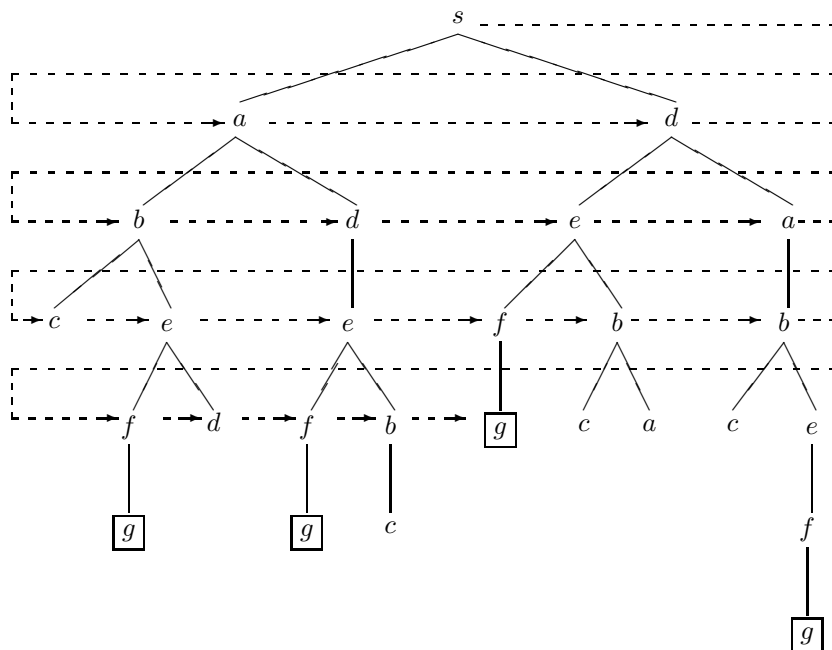


Figure 2.19: Breadth First

- Rename the loop predicate to *bfs_loop*,
- Change the order of the first two arguments in the *append* goal,
- Leave the definition of the auxiliary predicates unchanged.

The behaviour of *breadth_first/4* is shown in Fig. 2.21. The same paths are found as before, albeit in a different order.

Exercise 2.6. Rewrite the definition of *breadth_first/4* in Fig. 2.20 using difference lists. Compare the performance of your solution with that of the old version.

Hints. You may take the model solution of Exercise 2.5, p. 175, or your own solution, and make the necessary changes: rename the loop predicate; modify the updating of the agenda (now represented as a difference list); and, use *extend_path_dl/3* as defined in the solution of Exercise 2.5. For later reference, the new version should be placed in the same file as the earlier, list based version (i.e. *bf.pl*). ■

2.5 Bounded Depth First Search

Analysing Depth First and Breadth First will show that (e.g. [29]), *on average*, to find a goal node,

- Depth First needs less computer memory than Breadth First,
- The time requirement of Breadth First is asymptotically comparable to that of Depth First, and,

```

:- use_module(links).
:- use_module(searchinfo).

breadth_first(Start,G_Pred,C_Pred,PathFound) :-
    bfs_loop([[Start]],G_Pred,C_Pred,PathFoundRev),
    reverse(PathFoundRev,PathFound).

bfs_loop([Path|_],G_Pred,_,Path) :- call(G_Pred,Path).
bfs_loop([[CurrNode|T]|Others],G_Pred,C_Pred,PathFound) :-
    successors(C_Pred,CurrNode,SuccNodes),
    findall(Node,(member(Node,SuccNodes),
        is_path([Node,CurrNode|T])),Nodes),
    extend_path(Nodes,[CurrNode|T],Paths),
    append(Others,Paths,NewOpenPaths),
    bfs_loop(NewOpenPaths,G_Pred,C_Pred,PathFound).

% auxiliary predicates ...
...

```

Modified Goal
(see Fig. 2.15, p. 65)

Copy from Fig. 2.15, p. 65

Figure 2.20: The File `bf.pl` – Breadth First with Path Checking

```

?- consult(bf).
% links compiled into edges 0.00 sec, 1,900 bytes
% searchinfo compiled into info 0.00 sec, 1,016 bytes
% bf compiled 0.05 sec, 4,948 bytes
Yes
?- breadth_first(s,goal_path,link,Path).
Path = [s, d, e, f, g] ;
Path = [s, a, b, e, f, g] ;
Path = [s, a, d, e, f, g] ;
Path = [s, d, a, b, e, f, g] ;
No

```

Figure 2.21: Interactive Session for `breadth_first/4`

- Breadth First always finds the *shortest* path to the goal node (if there is one) whereas (for infinite search trees) Depth First may fail to find a goal node even if one exists.

Bounded Depth First search, shown in Algorithm 2.5.1, p. 71, combines the idea of the two search algorithms: it will explore the search tree up to a specified depth (the *horizon*) by Depth First. Bounded Depth First is also the basis for the more sophisticated *Iterative Deepening*, to be discussed in the next section.

Algorithm 2.5.1: BOUNDED_DF(*StartNode*, *G_Pred*, *C_Pred*, *Horizon*)

comment: Bounded Depth First Search.
Procedures are assumed available for

- Testing whether a path is a goal path by using the procedure *in G_Pred*;
- Finding successors of a node by using the connectivity procedure *in C_Pred*.

procedure ISPATH(*list*)
comment: Returns a Boolean value.
Is application specific.

⋮

main
RootNode ← *StartNode*
OpenPaths ← [[*RootNode*]]
[[*H|T*]|*TailOpenPaths*] ← *OpenPaths*
ListLength ← LENGTH([*H|T*])
PathLength ← *ListLength* − 1
while [*H|T*] is not a goal path
 if *PathLength* < *Horizon*
 then { *SuccList* ← successors of *H*
 ONodes ← list of *S* ∈ *SuccList* with
 ISPATH([*S*, *H|T*])
 NewPaths ← EXTENDPATH(*ONodes*, [*H|T*])
 else { *NewPaths* ← []
 do { *OpenPaths* ← *NewPaths* ++ *TailOpenPaths*
 if *OpenPaths* = []
 then return (*failure*)
 [[*H|T*]|*TailOpenPaths*] ← *OpenPaths*
 ListLength ← LENGTH([*H|T*])
 PathLength ← *ListLength* − 1
 Path ← REVERSE([*H|T*])
output (*Path*)

Exercise 2.7. In the query below, the predicate *bounded_df/5* is used to search the tree in Fig. 2.5 up to level 5 for the goal node *g*.

```
?- bounded_df(s,goal_path,link,5,PathFound).
PathFound = [s, a, b, e, f, g] ;
PathFound = [s, a, d, e, f, g] ;
PathFound = [s, d, e, f, g] ;
```

```

:- module(bounded_depth_first,[bounded_df/5]).7
:- use_module(links).
:- use_module(searchinfo).

bounded_df(Start,G_Pred,C_Pred,Horizon,PathFound) :-
    b_dfs_loop([[Start]],G_Pred,C_Pred,Horizon,PathFoundRev),
    reverse(PathFoundRev,PathFound).

...
% auxiliary predicates ...
...

```

Diagram annotations:

- A right curly brace groups the three lines of code above the `% auxiliary predicates ...` line. An arrow points from this brace to a box containing the text: *Loop Predicate b_dfs_loop/5 to be defined here*.
- A right curly brace groups the two lines of code below the `% auxiliary predicates ...` line. An arrow points from this brace to a box containing the text: *Copy from Fig. 2.15, p. 65*.

Figure 2.22: The File `bdf.pl` – Bounded Depth First (for Exercise 2.7)

No

Based on Algorithm 2.5.1, define `bounded_df/5` by completing the missing parts in Fig. 2.22.

Hint. The definition of `b_dfs_loop/5` may be obtained from that of `dfs_loop/4` in Fig. 2.15 by augmenting the latter with a new argument for the *horizon*. ■

2.6 Iterative Deepening

Bounded Depth First search is invoked here repeatedly with a successively larger horizon. This may be performed until a path to the goal node is found or until some CPU time limit is exceeded. We choose the former with unit increment. An implementation and a test run are shown in Figs. 2.23 and 2.24, respectively.⁸ Iterative Deepening may seem computationally wasteful as at any one stage the previous stage is recomputed but it can be shown that it is asymptotically optimal (eg [29]).

Exercise 2.8. The interactive session in Fig. 2.24 illustrates that, on backtracking, Iterative Deepening will rediscover the goal paths found earlier. Modify our implementation of Iterative Deepening such that this does not happen, i.e. paths found earlier for a smaller horizon should be ignored.

Hint. Fig. 2.25 shows a sample session with this modified version. The previous horizon is recorded in the database by means of the predicate `lastdepth/1`. Goal paths shorter than the value herein are ignored. To implement this, you will have to modify the first clause of `b_dfs_loop/5` in `bdf.pl`. You will also have to arrange for the updating of `lastdepth/1` in the database. ■

Exercise 2.9. Yet another, and perhaps the most usual form of Iterative Deepening will find the (leftmost) goal node at the shallowest depth (presuming that one exists) and then stop searching. For our example, such a version will respond as follows,

```
?- iterative_deepening(s,goal_path,link,PathFound).
```

⁷The predicate `bounded_df/5` is declared public because it will be used later in another module (see Sect. 2.6).

⁸The notes in Fig. 2.24 concerning the *horizon* refer to Fig 2.5, p. 51.

```
:- use_module(bdf).

iterative_deepening(Start,G_Pred,C_Pred,PathFound) :-
    iterative_deepening_aux(1,Start,G_Pred,C_Pred,PathFound).

iterative_deepening_aux(Depth,Start,G_Pred,C_Pred,PathFound) :-
    bounded_df(Start,G_Pred,C_Pred,Depth,PathFound).
iterative_deepening_aux(Depth,Start,G_Pred,C_Pred,PathFound) :-
    NewDepth is Depth + 1,
    iterative_deepening_aux(NewDepth,Start,G_Pred,C_Pred,PathFound).
```

Figure 2.23: The File `iterd.pl` – Iterative Deepening

```
PathFound = [s, d, e, f, g] ;
No
```

```

?- consult(iterd).
% links compiled into edges 0.06 sec, 1,856 bytes
% searchinfo compiled into info 0.00 sec, 1,016 bytes
% bdf compiled into bounded_depth_first 0.06 sec, 5,784 bytes
% iterd compiled 0.06 sec, 7,664 bytes
Yes
?- iterative_deepening(s,goal_path,link,PathFound).
PathFound = [s, d, e, f, g] ;
PathFound = [s, a, b, e, f, g] ;
PathFound = [s, a, d, e, f, g] ;
PathFound = [s, d, e, f, g] ;
PathFound = [s, a, b, e, f, g] ;
PathFound = [s, a, d, e, f, g] ;
PathFound = [s, d, e, f, g] ;
PathFound = [s, d, a, b, e, f, g] ;
PathFound = [s, a, b, e, f, g] ;
...

```

Figure 2.24: Sample Session – Iterative Deepening

Implement this version of Iterative Deepening. ■

Finally, notice that, for finite search trees, Iterative Deepening has an unpleasant feature not found with the other blind search algorithms: if there is no goal node, Iterative Deepening won't terminate.⁹ This will cause problems in applications where a sequence of *potential* start nodes is supplied to the algorithm some of which won't lead to a goal node. (An example of this will be seen in Sect. 2.8).

2.7 The Module *blindsearches*

The implementations of the algorithms from the preceding sections have been put together in `blindsearches.pl` to form the module *blindsearches*. This allows us to create an implementation of the network search problem anew which then may serve as a template for other uses of *blindsearches*. The top level is `netsearch.pl`, Fig. 2.26, p. 75. The following shows an interactive session using *search/0* from `netsearch.pl`.

```

?- consult(netsearch).
% links compiled into edges 0.00 sec, 1,900 bytes
% searchinfo compiled into info 0.00 sec, 1,016 bytes
% blindsearches compiled into blindsearches 0.06 sec, 7,284 bytes
% netsearch compiled 0.06 sec, 14,312 bytes
?- search.
Enter start state (a/b/c/d/e/f/s)... s.
Select algorithm (df/df_dl/bf/bf_dl/bdf/id)... bdf.

```

⁹For example, if we apply the query

```
?- iterative_deepening(u,goal_path,link,PathFound).
```

with the database in `links.pl` (as augmented in Exercise 2.2, p. 59), we won't get any response.

```

?- iterative_deepening(s,goal_path,link,PathFound).
PathFound = [s, d, e, f, g] ;
PathFound = [s, a, b, e, f, g] ;
PathFound = [s, a, d, e, f, g] ;
PathFound = [s, d, a, b, e, f, g] ; } ← No response after this
Ctrl+C
Action (h for help) ? abort
% Execution Aborted
?- lastdepth(D).
D = 396 } ← Last value of horizon
Yes

```

Figure 2.25: Sample Session – Modified Iterative Deepening (for Exercise 2.8)

```

:- use_module(links).
:- use_module(searchinfo).
:- use_module(blindsearches).

search :-
    G = goal_path,
    get_start_state(S),
    select_algorithm(A),
    (A = bdf, get_horizon(Horizon); true), !,
    ((A = df, depth_first(S,G,link,PathFound));
     (A = df_dl, depth_first_dl(S,G,link,PathFound));
     (A = bf, breadth_first(S,G,link,PathFound));
     (A = bf_dl, breadth_first_dl(S,G,link,PathFound));
     (A = bdf, bounded_df(S,G,link,Horizon,PathFound));
     (A = id, iterative_deepening(S,G,link,PathFound))),
    show_nodes(PathFound),
    terminate.

% missing predicates (shaded) to be defined here ...
...

```

Figure 2.26: The File netsearch.pl (for Exercise 2.10)

```
Enter horizon... 5.  
Nodes visited: s -> a -> b -> e -> f -> g  
Stop search? (y/n) n.  
Nodes visited: s -> a -> d -> e -> f -> g  
Stop search? (y/n) y.  
Yes
```

Exercise 2.10. Define the missing predicates (shaded) in Fig. 2.26. (You will have to use the built-in predicate *read/1* for reading a term. Notice that the input from the keyboard always finishes with a dot (.) as shown above.) ■

2.8 Application: A Loop Puzzle

2.8.1 The Puzzle

This is a more substantial example showing that some problems can be formulated as a network search problem thereby making them amenable to a solution by the algorithms described earlier. The idea of the puzzle considered here originates from the puzzle magazine [17].

We are given a rectangular board some positions of which are marked by circles (0) and sharps (#) as shown in the upper half of Fig. 2.27, p. 78. The task is to place a closed *loop* of a rope onto the board such that the following conditions are met:

- The rope connects contiguous positions horizontally or vertically but not diagonally. It does not self-intersect.
- Each position is visited by the rope at most once. (This follows, of course, also from the fact that the rope is not self-intersecting.) In particular, there may well be positions which are not visited at all.
- Each marked position is visited exactly once.
- Adjacent marks on the rope of the *like* kind (i.e. both circles or both sharps) are connected by a straight piece of rope.
- Adjacent marks on the rope which are *different* (i.e. if one is a circle and the other is a sharp) are connected by a piece of rope which takes a right angle turn.

A puzzle from [17] is solved in Fig. 2.27 by the model implementation. It is run interactively and carries out the following steps in turn:

1. It displays a sketch of the board and the arrangement of the marks (circles and sharps).
2. It gives the user a choice between the various search algorithms.
3. It tries to solve the problem and, if a solution exists, it gives a pictorial display of the loop's position on the board.¹⁰ If no solution is found, *loop/0* should fail. Furthermore, if there are several solutions, the implementation should find all of them.

2.8.2 A 'Hand-Knit' Solution

The core question is obviously how the present problem translates to a network search problem. (For the time being, we won't be concerned with the generation of the interface and display of the loop found as they are relatively straightforward, though laborious.)

As a first step, we want to illustrate by way of the specific case from Sect. 2.8.1 how the problem can be solved by directly creating (i.e. defining by facts) the predicates needed by the module *blindsearches*. The information concerning the specifics of the puzzle is defined in the file *loop_puzzle1.pl* shown in Fig. 2.28. Before defining the connectivity predicate which, as usual, will be called *link/2*, we will have to find a suitable representation for the system's states. The rope will be pieced together segment by segment, i.e. by progressing from one mark to the next. It seems therefore appropriate to identify the states of the system (i.e. the *nodes* of the corresponding network) with rope *segments* connecting marked positions.

A list representation will be used for rope segments and progression in the list will be from right to left. Thus, for example, movement from a circle at position *pos(1,4)* to a sharp at position *pos(2,2)* is indicated by either of the following two segments.

$$[\text{pos}(2,2), \text{pos}(2,3), \text{pos}(2,4)] \quad (2.1)$$

¹⁰A solution may be missed, however, if Bounded Depth First search is used. Furthermore, if Iterative Deepening is selected in our implementation, it will not terminate if the internally attempted start state does not lead to a solution.

```

?- consult(loop_puzzle1).
% blindsearches compiled into blindsearches 0.00 sec, 7,284 bytes
% small_board compiled into small_board 0.00 sec, 6,224 bytes
% board compiled into board 0.05 sec, 7,696 bytes
% loops compiled into loops 0.11 sec, 31,028 bytes
% loop_puzzle1 compiled 0.11 sec, 32,324 bytes
Yes
?- loop.
+---+---+---+---+---+
| | | | 0 | | # | 1
+---+---+---+---+---+
| # | # | | | | | 2
+---+---+---+---+---+
| | | | | 0 | | 3
+---+---+---+---+---+
| # | 0 | | | | | 4
+---+---+---+---+---+
| | | | | # | | 5
+---+---+---+---+---+
| | | | | 0 | | 6
+---+---+---+---+---+
  1   2   3   4   5   6

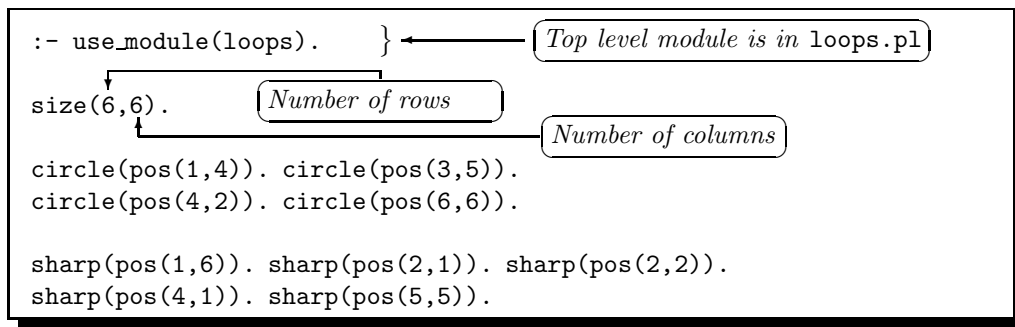
Select algorithm (df/df_dl/bf/bf_dl/bdf/id)... df.

+---+---+---+---+---+
| *****0 *****# |
| * | * | * | * | * |
+---+---+---+---+---+
| * | * | * | * | * |
| # | #***** | * | * |
| * | * | * | * | * |
+---+---+---+---+---+
| * | * | * | * | * |
| * | *****0 | * | * |
| * | * | * | * | * |
+---+---+---+---+---+
| * | * | * | * | * |
| # | 0***** | * | * |
| * | * | * | * | * |
+---+---+---+---+---+
| * | * | * | * | * |
| * | *****#***** |
| * | * | * | * | * |
+---+---+---+---+---+
| * | * | * | * | * |
| *****0 | * | * |
| * | * | * | * | * |
+---+---+---+---+---+

Stop search? (y/n) y.
Yes

```

Figure 2.27: Sample Session – The Loop Puzzle

Figure 2.28: The File `loop_puzzle1.pl`

and

$$[\text{pos}(2,2), \text{pos}(1,2), \text{pos}(1,3)] \quad (2.2)$$

These segments are indicated by solid arrows in Fig. 2.29. Notice that the position *at* which the segment arrives, here `pos(2,2)`, features as the head of its list representation whereas the board position *from* which the segment originates is omitted from the list.

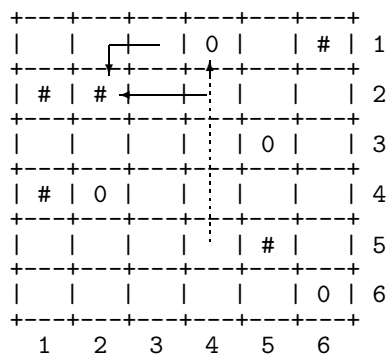


Figure 2.29: Constructing a Solution of the Loop Puzzle

(This will enable us simply to string together the final rope from its segments without being concerned with duplication of some positions.) The marked positions connected by a segment will be *adjacent* on the rope of which the segment is part of. We require therefore that the only marked position be the head of the segment's list representation. Thus, for example,

$$[\text{pos}(4,1), \text{pos}(3,1), \text{pos}(2,1), \text{pos}(1,1), \text{pos}(1,2), \text{pos}(1,3)]$$

is *not* a segment as it meets the marked position $\text{pos}(2,1)$ 'on its way' from $\text{pos}(1,4)$ to $\text{pos}(4,1)$. We now take the segment

$$[\text{pos}(1,4), \text{pos}(2,4), \text{pos}(3,4), \text{pos}(4,4), \text{pos}(5,4)] \quad (2.3)$$

which is deemed to stretch from the sharp at $\text{pos}(5,5)$ to the circle at $\text{pos}(1,4)$. This is indicated by the dashed arrow in Fig. 2.29. (The other potential segment connecting the same positions as the one in (2.3) must be ruled out since it is blocked by the mark (circle) in $\text{pos}(3,5)$.) To indicate that the segment in (2.2) is linked to that in (2.3), we declare in the database the following fact:

$$\text{link}([\text{pos}(1,4), \text{pos}(2,4), \text{pos}(3,4), \text{pos}(4,4), \text{pos}(5,4)], [\text{pos}(2,2), \text{pos}(1,2), \text{pos}(1,3)]).$$

Notice that the *order* of the arguments in *link/2* matters: according to our interpretation, the segment in the first argument is visited first, followed by the segment in the second argument. The corresponding fact linking the segments in (2.3) and (2.1) does not hold if self-intersecting loops are excluded. Let us assume, however, that *at this stage* we do not care whether a rope is self-intersecting since this will be attended to later when we define the predicate *is-path/1*. Then, a more concise and more general form of the above fact is given by

$$\text{link}([\text{pos}(1,4)|_], [\text{pos}(2,2), \text{pos}(1,2), \text{pos}(1,3)]).$$

(This simply states that the segment $[\text{pos}(2,2), \text{pos}(1,2), \text{pos}(1,3)]$ will join *any* segment pointing at $\text{pos}(1,4)$.) There are three other segments also originating *from* the circle in $\text{pos}(1,4)$; they give rise to the following fact each.

$$\begin{aligned} &\text{link}([\text{pos}(1,4)|_], [\text{pos}(2,1), \text{pos}(1,1), \text{pos}(1,2), \text{pos}(1,3)]). \\ &\text{link}([\text{pos}(1,4)|_], [\text{pos}(2,2), \text{pos}(2,3), \text{pos}(2,4)]). \\ &\text{link}([\text{pos}(1,4)|_], [\text{pos}(5,5), \text{pos}(5,4), \text{pos}(4,4), \text{pos}(3,4), \text{pos}(2,4)]). \end{aligned}$$

```

:- use_module(blindsearches).

...
start_state([pos(2,1),pos(1,1),pos(1,2),pos(1,3)]).

goal_path([H|T]) :- length([H|T],9),
                    last(E,T),
                    link(H,E).

is_path([H|T]) :- not(prohibit([H|T])).

prohibit([S|[H|_]]) :- not(disjoint(S,H)).
prohibit([S|[_|T]]) :- prohibit([S|T]).

disjoint([],_).
disjoint([H|T],S) :- not(member(H,S)), disjoint(T,S).

```

Figure 2.30 shows annotations for the Prolog code in `hand_knit.pl`:

- A bracket points to the `link/2` predicate definition with the note: *Define link/2 here (see Exercise 2.11, p. 81)*
- A bracket points to the `length([H|T],9)` and `last(E,T)` goals with the note: *The goal path has 9 segments*
- A bracket points to the `link(H,E)` goal with the note: *The goal path is closed*
- A bracket points to the `prohibit([S|[H|_]]) :- not(disjoint(S,H)).` and `prohibit([S|[_|T]]) :- prohibit([S|T]).` rules with the note: *Exclude self-intersecting paths*

Figure 2.30: The File `hand_knit.pl`

In a similar fashion, the segments originating *from* the circle in `pos(3,5)` give rise to the facts

```

link([pos(3,5)|_], [pos(1,6),pos(2,6),pos(3,6)]).
link([pos(3,5)|_], [pos(1,6),pos(1,5),pos(2,5)]).
link([pos(3,5)|_], [pos(2,1),pos(3,1),pos(3,2),pos(3,3),pos(3,4)]).
link([pos(3,5)|_], [pos(2,2),pos(3,2),pos(3,3),pos(3,4)]).
link([pos(3,5)|_], [pos(2,2),pos(2,3),pos(2,4),pos(2,5)]).
link([pos(3,5)|_], [pos(4,1),pos(3,1),pos(3,2),pos(3,3),pos(3,4)]).

```

Exercise 2.11. Complete the definition of `link/2` in this fashion. There will be 37 facts in total forming 9 groups, each group corresponding to a marked position. (You will find the solution of this exercise in the file `hand_knit.pl`.) ■

The definition of `link/2` and those of some other predicates¹¹ are in the file `hand_knit.pl`, partially shown in Fig. 2.30. It is also seen from `hand_knit.pl` that one of the segments has been chosen as a start state by visual inspection of Fig. 2.29.¹² We are now in a position to find a solution *interactively*. After consulting `hand_knit.pl`, we invoke `depth_first/4` as follows.

```

?- start_state(_S), depth_first(_S,goal_path,link,_PathFound),
   write_term(_PathFound,[]).
[[pos(2, 1), pos(1, 1), pos(1, 2), pos(1, 3)],
 [pos(4, 1), pos(3, 1)],
 [pos(6, 6), pos(6, 5), pos(6, 4), pos(6, 3), pos(6, 2), pos(6, 1), pos(5, 1)],
 [pos(5, 5), pos(5, 6)],

```

¹¹ Notice that the predicate `is_path/1` in `hand_knit.pl` is ‘visible’ from the module `blindsearches` without it being exported.

¹² A reasoned way to get hold of a start state is as follows. Pick *any* marked position and try out all segments originating from it. If there is a solution to the problem, then at least one of the segments thus produced may serve as a start state since the rope must pass through this position in particular.

```
[pos(4, 2), pos(5, 2), pos(5, 3), pos(5, 4)],
[pos(1, 6), pos(2, 6), pos(3, 6), pos(4, 6), pos(4, 5), pos(4, 4), pos(4, 3)],
[pos(3, 5), pos(2, 5), pos(1, 5)],
[pos(2, 2), pos(3, 2), pos(3, 3), pos(3, 4)],
[pos(1, 4), pos(2, 4), pos(2, 3)]]
```

A list comprising 9 path segments has been returned. It is to be read from left to right but the list representations of the segments are read from right to left. It is perhaps easier to interpret the result if we subsequently reverse this list and then flatten it. The list thus produced will be a right-to-left display of the positions visited.

```
?- start_state(_S), depth_first(_S, goal_path, link, _PathFound),
    reverse(_PathFound, _R), flatten(_R, _F), write_term(_F, []).
[pos(1, 4), pos(2, 4), pos(2, 3), pos(2, 2), pos(3, 2), pos(3, 3),
 pos(3, 4), pos(3, 5), pos(2, 5), pos(1, 5), pos(1, 6), pos(2, 6),
 pos(3, 6), pos(4, 6), pos(4, 5), pos(4, 4), pos(4, 3), pos(4, 2),
 pos(5, 2), pos(5, 3), pos(5, 4), pos(5, 5), pos(5, 6), pos(6, 6),
 pos(6, 5), pos(6, 4), pos(6, 3), pos(6, 2), pos(6, 1), pos(5, 1),
 pos(4, 1), pos(3, 1), pos(2, 1), pos(1, 1), pos(1, 2), pos(1, 3)]
```

```

:- use_module(blindsearches).
:- use_module(automated).

size(6,6).

circle(pos(1,4)). circle(pos(3,5)).
circle(pos(4,2)). circle(pos(6,6)).

sharp(pos(1,6)). sharp(pos(2,1)).
sharp(pos(2,2)). sharp(pos(4,1)). sharp(pos(5,5)).

```

Figure 2.31: The File `loop_puzzle1a.pl`

The path thus obtained is seen to be the one shown in Fig. 2.27.¹³

2.8.3 Project: Automating the Solution Process

In the ‘hand-knit’ solution from the previous section, the information specific to the puzzle was conveyed to Prolog via the predicate `link/2`, defined in `hand_knit.pl` by Prolog *facts* which were arrived at laboriously by visual inspection of `loop_puzzle1.pl`. This arrangement, though unsatisfactory, has been useful in showing that this type of puzzle can be solved as a network search problem. We are aiming for a more flexible and automated implementation, however, which will solve *any* problem of this type by combining the problem-specific information from a file like `loop_puzzle1.pl` with a rule-based and *not problem-dependent* definition of `link/2`.¹⁴

You will be asked to find a rule-based definition of `link/2` in Exercise 2.12 below. The suggested *file structure* is as follows. The information concerning *this particular* puzzle should be recorded in the file `loop_puzzle1a.pl`¹⁵ as shown in Fig 2.31, p. 83. All the other predicates pertinent to this *type* of puzzle should be defined in the file `automated.pl` as outlined in Fig. 2.32, p. 84.

Exercise 2.12. To get a semi-automated solution¹⁶ of the loop puzzle as indicated by the interactive session in Fig. 2.35, p. 88, augment the file `hand_knit.pl` by defining `link/2` by *rules*. The augmented file will be the first version of `automated.pl`. Below you will find some guidance on the implementation of `link/2`. ■

Implementing `link/2`

At variance with the fact-based version of `link/2`, now linking intersecting segments will be disallowed. Thus, for example, whereas

¹³To obtain a *loop*, the positions `pos(1,4)` and `pos(1,3)` have been joined since they are the two extreme entries (first and last) of the path found.

¹⁴Another approach more in tune with Sect. 2.8.2 will first create in the database at runtime the problem-specific facts defining `link/2`. (Alternatively, a problem-specific (temporary) file akin to `hand_knit.pl` may be created and consulted at runtime.) This should be accomplished by a second order predicate reading the definitions of `size/2`, `circle/1` and `sharp/1` from `loop_puzzle1.pl` (or its analogue). Subsequently, run the search as in Sect. 2.8.2.

¹⁵The suffix ‘a’ in the filename indicates that the solution process is *automated*.

¹⁶The initial segment is supplied via `start_state/1` by *manual* input. A fully automated solution is considered in Exercise 2.13.

```

:- module(auto,[link/2,maybe_start_state/1,goal_path/1,is_path/1]).

...
}
...
}
goal_path([H|T]) :- number_of_marks(M),
                    length([H|T],M),
                    last(E,T),
                    link(H,E).
...
}
is_path([H|T]) :- not(prohibit([H|T])).
prohibit([S|[H|_]]) :- not(disjoint(S,H)).
prohibit([S|[_|T]]) :- prohibit([S|T]).
disjoint([],_).
disjoint([H|T],S) :- not(member(H,S)),
                    disjoint(T,S).

```

For Exercise 2.13 only

Define *link*/2 here (see Exercise 2.12)

Define *maybe_start_state*/1 here (see Exercise 2.13)

Modified definition of *goal_path*/1 (see Exercise 2.13)

Define *number_of_marks*/1 here (see Exercise 2.13)

Copy from *hand_knit.pl* (see Fig. 2.30)

Figure 2.32: The File *automated.pl*

```
link([pos(5,5),pos(4,5),pos(4,4),pos(4,3)],
     [pos(1,4),pos(2,4),pos(3,4),pos(4,4),pos(5,4)]).
```

follows from the definition of *link/2* in *hand_knit.pl*, it cannot be inferred by our rule-based version of *link/2* in *automated.pl*:

```
?- link([pos(5,5),pos(4,5),pos(4,4),pos(4,3)],S).
S = [pos(4, 2), pos(5, 2), pos(5, 3), pos(5, 4)] ;
S = [pos(6, 6), pos(5, 6)] ;
S = [pos(6, 6), pos(6, 5)] ;
No
```

Does it matter if this additional condition is imposed? No, the final result won't be affected as paths containing self-intersecting linked segments are themselves self-intersecting and will therefore be disallowed by *is_path/1*. However, whereas *link/2* was previously defined by a relatively small number of facts, the resulting network is more complex. It will be seen that the imposed condition is easily incorporated in the definition of *link/2* and, as indicated above, it should give rise to a simpler network, i.e. to a one with a lesser number of connections. (You will be asked to compare the two networks as part of Exercise 2.14, p. 87.)

The dashed arrows in Fig. 2.33 stand for segments connected to *[pos(5,5),pos(4,5),pos(4,4),pos(4,3)]* which itself is shown as a continuous arrow. We require furthermore that

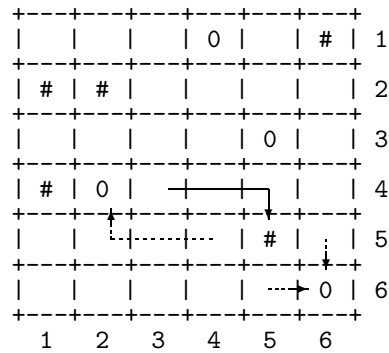


Figure 2.33: Constructing a Loop

- *link/2* should fail if the first argument is not unified with a valid segment:

```
?- link([pos(5,3),pos(6,3),pos(6,4),pos(6,5)], S).
No
```

- *link/2* should also fail if the arguments are unified with valid segments, that, however, are not linked:

```
?- link([pos(6,6),pos(5,6)], [pos(6,6),pos(6,5)]).
No
```

- *link/2* should succeed if the arguments are unified with linked segments:

```
?- link([pos(2,1),pos(3,1)], [pos(2,2)]).
Yes
```

We now want to indicate how *link/2* should be defined. Let us assume that two marked positions of the like kind should be linked. This will be accomplished by the clause

```
link([Pos1/T1],[Pos2/T2]) :- ((circle(Pos1), circle(Pos2)); (sharp(Pos1), sharp(Pos2))),
    straight(Pos1,[Pos2/T2],Pos2),
    not((member(Pos,T2),(circle(Pos);sharp(Pos)))),
    disjoint([Pos1/T1],[Pos2/T2]).
```

where the auxiliary predicate *straight(+P1, ?S, +P2)* connects any two positions *P1* and *P2* sharing the same row or column; details of what is required may be gleaned from the query below.

```
?- auto:straight(pos(3,4),S,pos(3,8)).
S = [pos(3, 8), pos(3, 7), pos(3, 6), pos(3, 5)]
?- auto:straight(pos(8,3),S,pos(4,3)).
S = [pos(4, 3), pos(5, 3), pos(6, 3), pos(7, 3)]
```

(We use the prefix *auto* in the above query as *straight/3* is not visible from outside the module *auto*.)

You are recommended to use the built-in predicates *bagof/3*, *between/3* and *reverse/2* in your definition of *straight/3*.

The corresponding clause of *link/2* for linking marked positions of an unlike kind uses the auxiliary predicate *turn(+P1, ?R, +P2)* where the positions *P1* and *P2* (not sharing the same row or column) are linked by the list *R* taking a right angle turn; for example,

```

?- consult(loop_puzzle1a).
...
?- maybe_start_state(_S), depth_first(_S,goal_path,link,_PathFound),
   reverse(_PathFound,_R), flatten(_R,_F), write_term(_F,[ ]).
[pos(1, 4), ..., pos(1, 3)]
Yes

```

Figure 2.34: Running the Automated Implementation of the Loop Puzzle

```

?- auto:turn(pos(6,4),R,pos(4,1)).
R = [pos(4, 1), pos(5, 1), pos(6, 1), pos(6, 2), pos(6, 3)]
?- auto:turn(pos(8,3),R,pos(4,2)).17
R = [pos(4, 2), pos(5, 2), pos(6, 2), pos(7, 2), pos(8, 2)]

```

To define *turn/3*, use *straight/3* and *append/3*.

Fully Automated Implementation

Exercise 2.13. To get an automated solution of the loop puzzle as indicated by the interactive session in Fig. 2.34, now augment the file `automated.pl` as follows.

- Define the predicate *maybe_start_state/1*, and make it a visible predicate by augmenting the *module* directive as indicated in Fig. 2.32. It should return on backtracking all segments emanating from an arbitrary but fixed marked position. As explained in footnote 12, p. 81, one of the segments returned by *maybe_start_state/1* will form part of the loop we are looking for.
- Define the predicate *number_of_marks/1* and modify the definition of *goal_path/1* as indicated in Fig. 2.32.

■

Exercise 2.14. (This exercise explores the idea mentioned in footnote 14, p. 83.) The ‘hand-knit’ solution outlined in Sect. 2.8.2 involved a *manual* implementation of *link/2* by defining it by Prolog *facts*. These facts were, of course, specific to the puzzle to be solved. Having now defined *link/2* by rules not referring to the particulars of the puzzle at hand, we have been able to automate the solution process. An alternative closer to the original idea would be automatically to define in the database *link/2* by the facts applicable to the particular problem. Use *link/2* to define by *facts* an equivalent new link predicate and use it to solve the loop puzzle. Determine the number of nodes and the number of directed edges of the corresponding network. Determine these quantities also for the network associated with the ‘hand-knit’ solution (Sect. 2.8.2) to confirm that the latter is indeed more complex.

■

¹⁷The L-shaped segment degenerates here into a straight line since it connects positions in adjacent columns.

```
?- consult(loop_puzzle1a).
% blindsearches compiled into blindsearches 0.05 sec, 7,380 bytes
% automated compiled into auto 0.00 sec, 5,752 bytes
% loop_puzzle1a compiled 0.05 sec, 14,576 bytes
Yes
?- consult(user).
|: start_state([pos(2,1),pos(1,1),pos(1,2),pos(1,3)]).
|: Ctrl+D
% user compiled 34.11 sec, 388 bytes
Yes
?- start_state(_S), depth_first(_S,goal_path,link,_PathFound), reverse(_PathFound,_R), flatten(_R,_F),
   write_term(_F, []).
[pos(1, 4), pos(2, 4), pos(2, 3), pos(2, 2), pos(3, 2), pos(3, 3), pos(3, 4), pos(3, 5), pos(2, 5), pos(1, 5),
 pos(1, 6), pos(2, 6), pos(3, 6), pos(4, 6), pos(4, 5), pos(4, 4), pos(4, 3), pos(4, 2), pos(5, 2), pos(5, 3),
 pos(5, 4), pos(5, 5), pos(5, 6), pos(6, 6), pos(6, 5), pos(6, 4), pos(6, 3), pos(6, 2), pos(6, 1), pos(5, 1),
 pos(4, 1), pos(3, 1), pos(2, 1), pos(1, 1), pos(1, 2), pos(1, 3)]
Yes
```

} ← Manual definition
of start_state/1

Figure 2.35: Semi-Automated Solution of the Loop Puzzle

```

?- consult([loop_puzzle1a, small_board]).
...
% loop_puzzle1a compiled 0.05 sec, 15,076 bytes
% small_board compiled into small_board 0.06 sec, 6,216 bytes
Yes
?- size(_Row,_Col), bagof(_C,circle(_C),_Cs),
   bagof(_S,sharp(_S),_Ss),
   make_small_board(_Row,_Col,_Cs,_Ss,_Board),
   disp_board(_Board).

```

			0		#	1
	#	#				2
				0		3
	#	0				4
					#	5
					0	6
1	2	3	4	5	6	

Yes

size/2, circle/1 and sharp/1 to be taken from loop_puzzle1a.pl (see Fig. 2.31, p. 83)

Figure 2.36: Session for Displaying the Board

2.8.4 Project: Displaying the Board

Exercise 2.15. To display the *marks' position* on the board, define

- *make_small_board(+Row,+Col,+Circles,+Sharps,-Board)* for unifying *Board* with the list of lines to be displayed where each line itself is represented as a list of one-character atoms; and,
- *disp_board(+Board)* for displaying *Board* on the terminal.

Fig. 2.36 shows how these predicates should behave. (The model solution is in `small_board.pl`.) ■

Exercise 2.16. To display a *path* on the board, define

- *make_board(+Row,+Col,+Path,-Board)* for creating a list-of-lists representation of *Board*, and,
- *show_board(+Board)* for displaying *Board* on the terminal.

Path is unified with a list of contiguous co-ordinate entries of the form *pos(...,...)*. Fig. 2.37 illustrates the point for a 2×5 board. (The model solution is in `board.pl`.) ■

```

?- consult(board).
% board compiled into board 0.00 sec, 8,216 bytes
?- make_board(2,5,[pos(1,1),pos(1,2),pos(2,2),pos(2,3),pos(2,4),pos(1,4),pos(1,5)],_Board),
   show_board(_Board).
+-----+-----+-----+-----+-----+
|      |      |      |      |      | |
|  *****  |      |      |      |      |
|      |  *  |      |      |  *  |      |
+-----+-----+-----+-----+-----+
|      |  *  |      |      |  *  |      |
|      |  *****  |      |      |      |
|      |      |      |      |      |
+-----+-----+-----+-----+-----+

```

Yes

Figure 2.37: Illustrating Exercise 2.16

```
?- make_board(2,5,[pos(1,1),pos(1,2),pos(2,2),pos(2,3),pos(2,4),pos(1,4),pos(1,5)],_Board),
   change_board('c',[pos(1,2),pos(1,4),pos(2,2),pos(2,4)],_Board,_NewBoard),
   show_board(_NewBoard).
```

```
+-----+-----+-----+-----+
|       |       |       |       |
|  *****c  |       |  c*****  |
|       |  *   |       |  *   |
+-----+-----+-----+-----+
|       |  *   |       |  *   |
|       |  c*****c  |       |
|       |       |       |       |
+-----+-----+-----+-----+
```

Yes

Figure 2.38: Illustrating Exercise 2.17

Exercise 2.17. Finally, for putting circles and sharps on the board, a predicate for writing a given character to specified positions on the board will be useful. This will be accomplished by *change_board/4* as illustrated in Fig. 2.38. (In the example we mark corner positions of the path with the character 'c'.) Define *change_board/4*. (The model solution is in `board.pl`.) ■

2.8.5 Complete Implementation

All the building blocks for solving the puzzle and displaying the loop found are now in place. In fact, this can be done interactively as shown in Fig. 2.39.

Exercise 2.18. It is very tedious to solve the loop puzzle interactively as shown in Fig. 2.39. Combine now the predicates from above to create a more user-friendly implementation which can be run as shown in Fig. 2.27, p. 78. You may model your implementation of the dialogue on that in `netsearch.pl` (see Fig. 2.26, p. 75). (For the model solution, see `loops.pl`.) ■

2.8.6 Full Board Coverage

Exercise 2.19. Suppose now that the specification is made somewhat stricter. In addition to the initial requirements we now also want every small square to be visited by the loop. You should modify your implementation to include this new feature.

Notes.

1. Whereas the earlier puzzle has a unique solution which happens to visit every position (even if we don't insist on this), the case shown in Fig. 2.40 (with the data in `loop_puzzle2.pl`) is more complex and will admit solutions of both kinds (Figs. 2.41 and 2.42). Use `loop_puzzle2.pl` for testing your solution.

Figure 2.39: Solving the Puzzle Interactively. (See Exercise 2.18.)

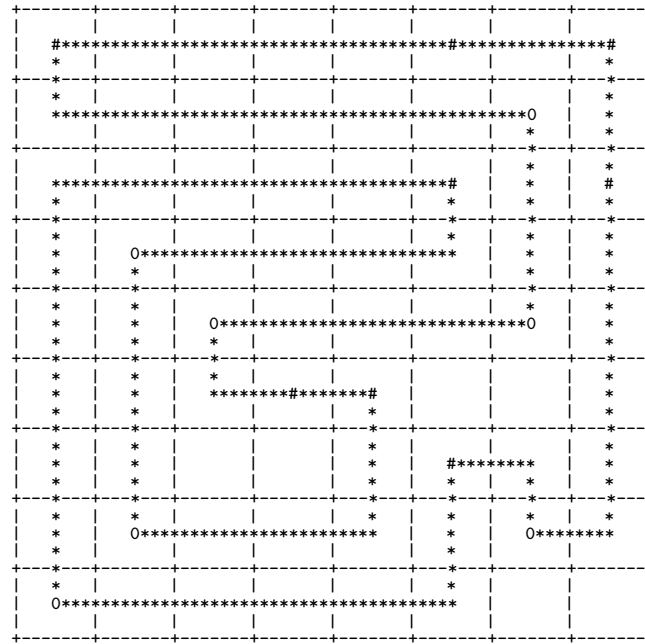


Figure 2.41: Some positions not visited

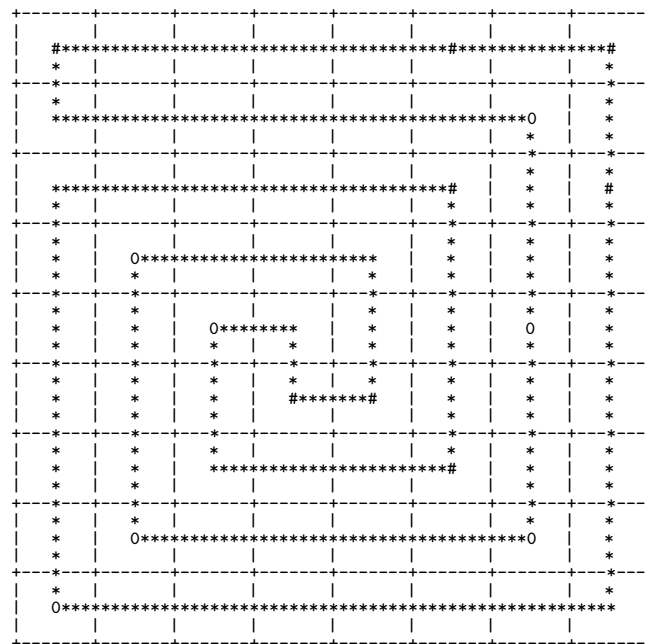


Figure 2.42: All positions visited

yet another starting segment giving rise to the same loop. For this version of the loop, the last segment will be `[pos(1,8), pos(2,8)]`, pointing at the position where the loop was mistakenly deemed to have started from. Situations such as this will be avoided if we stipulate that the head of the last segment be identical to the seed position; an augmented definition of *goal_path/1* to reflect this, is shown in (P-2.1).

Prolog Code P-2.1: Augmented definition of *goal_path/1*

```

1 goal_path([LastSegment|T]) :- number of marks(M),
2                               length([LastSegment|T],M),
3                               last(FirstSegment,T),
4                               link(LastSegment,FirstSegment),
5                               seed([SeedPosition]),           % added goal
6                               LastSegment = [SeedPosition|_]. % added goal

```

2.8.8 Variants of the Loop Puzzle

A Loop with ‘Kinks’

In this loop puzzle from [18], one symbol is used only, the circle (0) say, for marking some positions on a rectangular board. We are required to find a loop such that

- Each board position should be visited by the loop exactly once.
- Pairs of marks lying adjacent on the loop should be connected by L-shaped segments (which may be referred to as *kinks*).

An example from [18] is solved by the model implementation in Fig. 2.43.

Exercise 2.20. Write a Prolog solution for the above loop puzzle. It may be assumed that not all four corner positions are marked. (This assumption will allow a start state (i.e. an initial loop segment) to be ‘grown’ from this empty corner.) It may also be assumed that top and bottom rows, and leftmost and rightmost columns all contain at least one mark.

You may retain the structure of the earlier implementation. Use the modules *small_board* and *board* as before for displaying the positions of the marks and that of the loop. The puzzle specific source files for the model solution are *kinks.pl* and *kinks1.pl – kinks5.pl*. ■

A ‘Straight’ Loop

This puzzle originates from [16]. As before, one symbol is used only for marking some positions on a rectangular board, the circle (0), say. We want to find a loop such that

- Each board position is visited by the loop exactly once.
- Marked positions are traversed without a right angle turn; hence the attribute *straight*.

An example from [16] is solved by the model implementation in Fig. 2.44.

Exercise 2.21. Write a Prolog implementation for solving the above loop puzzle.

Hints.

1. In the model solution, all viable loop segments of length three form the system states; they may be denoted, for instance, by a term *state/3* with its arguments standing for three contiguous board positions. Given some state, the *link/2* predicate will generate all its children as shown in the queries below for the puzzle in Fig. 2.44.

```
?- link(state(pos(3,3),pos(2,3),pos(2,4)),S).
S = state(pos(3, 2), pos(3, 3), pos(2, 3)) ;
S = state(pos(3, 4), pos(3, 3), pos(2, 3)) ;
S = state(pos(4, 3), pos(3, 3), pos(2, 3)) ;
No
?- link(state(pos(3,4),pos(3,3),pos(2,3)),S).
S = state(pos(3, 5), pos(3, 4), pos(3, 3)) ;
No
```

It is seen that linked segments overlap by one position and that the *state/3* term can be thought of as a ‘window’ of size three progressing to the left. The second query above shows that the mark in *pos(3,4)* is traversed by a straight segment.

2. Because of the straightness condition, there can’t be any marks in the corners. We may therefore place the initial segment in the top left-hand corner.

The files *straightloop.pl* and *straightloop1.pl – straightloop3.pl* are the puzzle specific source for the model solution. ■

```
?- consult(kinks5).
% blindsearches compiled into blindsearches 0.00 sec, 7,312 bytes
% small_board compiled into small_board 0.00 sec, 6,224 bytes
% board compiled into board 0.00 sec, 7,696 bytes
% kinks compiled into kinks 0.00 sec, 34,736 bytes
% kinks5 compiled 0.10 sec, 36,480 bytes
```

Yes

```
?- loop.
```

```
+-----+
| | | | 0 | | | 0 | 1
+-----+
| | 0 | | | 0 | | | 2
+-----+
| | | | 0 | | 0 | | 3
+-----+
| | 0 | | | | | | 4
+-----+
| | | | | 0 | | 0 | 5
+-----+
| | | 0 | | | 0 | | 6
+-----+
| | | 0 | | 0 | | | 7
+-----+
| 0 | | | 0 | | | | 8
+-----+
1 2 3 4 5 6 7 8
```

For displaying the boards, use the modules *small_board* and *board* as specified in Sect. 2.8.4.

```
Select algorithm (df/df_dl/bf/bf_dl/bdf/id)... id.
```

```
+-----+
| *****0 | *****0 |
| * | * | * | * | * | * |
+-----+
| * | * | * | * | * | * |
| * | 0***** | 0***** |
| * | * | * | * | * | * |
+-----+
| * | * | * | * | * | * |
| * | *****0 | *****0 |
| * | * | * | * | * | * |
+-----+
| * | * | * | * | * | * |
| * | 0***** | * | * | * | * |
| * | * | * | * | * | * |
+-----+
| * | * | * | * | * | * |
| * | * | * | *****0 | *****0 |
| * | * | * | * | * | * |
+-----+
| * | * | * | * | * | * |
| * | * | 0***** | 0***** |
| * | * | * | * | * | * |
+-----+
| * | * | * | * | * | * |
| * | *****0 | *****0 | *****0 |
| * | * | * | * | * | * |
+-----+
| * | * | * | * | * | * |
| * | 0***** | 0***** |
| * | * | * | * | * | * |
+-----+
```

```
Stop search? (y/n) y.
```

Yes

Figure 2.43: Solving the Loop Puzzle – Variant One

```
%- consult(straightloop3).
% blindsearches compiled into blindsearches 0.00 sec, 7,328 bytes
% small_board compiled into small_board 0.00 sec, 6,224 bytes
% board compiled into board 0.00 sec, 7,712 bytes
% straightloop compiled into straightloop 0.00 sec, 30,048 bytes
% straightloop3 compiled 0.00 sec, 31,192 bytes
```

Yes

?- loop.

```
+---+---+---+---+---+
| | | | | | 1
+---+---+---+---+
| | 0 | | 0 | | 2
+---+---+---+---+
| 0 | | | 0 | | 3
+---+---+---+---+
| | | 0 | | | 4
+---+---+---+---+
| | | 0 | | 0 | 5
+---+---+---+---+
| | | | 0 | | 6
+---+---+---+---+
1 2 3 4 5 6
```

For displaying the boards, use the modules *small_board* and *board* as specified in Sect. 2.8.4.

Select algorithm (df/df_dl/bf/bf_dl/bdf/id)... id.

```
+---+---+---+---+---+
| ***** | ***** |
| *         | *         |
+---+---+---+---+
| *         | *         |
| * 0       | *****0***** |
| *         | *         |
+---+---+---+---+
| *         | *         |
| 0         | *****0***** |
| *         | *         |
+---+---+---+---+
| *         | *         |
| *         | *****0***** |
| *         | *         |
+---+---+---+---+
| *         | *         |
| *         | *****0***** |
| *         | *         |
+---+---+---+---+
| *         | *         |
| *****0***** |
| *         | *         |
+---+---+---+---+
```

Stop search? (y/n) y.

Yes

Figure 2.44: Solving the Loop Puzzle – Variant Two

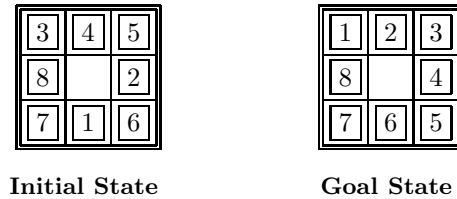


Figure 2.45: An Eight Puzzle

2.9 Application: The Eight Puzzle

2.9.1 The Puzzle

This is a standard example in AI and it is used for assessing the performance of search algorithms [27].

There are eight tiles, numbered 1 to 8, on a 3×3 board. The objective is to transform an initial tile arrangement into a given goal state; an example is shown in Fig. 2.45. In each transformation step, a new tile arrangement should be obtained by sliding a tile to the empty position.

The number of states of this puzzle is $9! = 362,880$. However, the state space is known to fall into two distinct components the states of each of which are mutually reachable from within but not from the other component's states. Below we show another popular choice for the goal state, residing in the other component.

1	2	3
4	5	6
7	8	

Alternative Goal State

Thus, if this latter arrangement is also admitted as a goal state, the puzzle will be solvable for any initial state.

2.9.2 Prolog Implementation

A sample run of the model implementation is shown in Fig. 2.46. The user may choose from eleven test cases; the first ten are from [15]. The test cases 1–10 are in order of increasing difficulty and are solvable with the goal state in Fig. 2.45. The eleventh test case is solvable for the alternative goal state.

Test Case Number		1	2	3	4	5	6	7	8	9	10
Goal Node at Depth		8	8	10	12	13	16	16	20	30	30
CPU Seconds	<i>bdf</i>	0.0	0.2	0.5	2.3	3.6	2.9	17.7	144.4	-	-
	<i>bf</i>	0.3	0.5	3.0	43.6	99.6	1523	-	-	-	-
	<i>id</i>	0.3	0.4	1.2	5.2	8.2	34.2	40.8	556.0	-	-

Table 2.1: CPU Times (in Seconds) for the Eight Puzzle with Blind Search

A summary of the results obtained on a 300 MHz PC is shown in Table 2.1: no entries are shown for unsuccessful runs due to stack overflow or prohibitively long computing times; and, the value chosen for the *horizon* in Bounded Depth First search is the minimum number of moves needed to reach the goal state (row two in Table 2.1).¹⁸

Implementation Details

The system's states are internally represented by the term *state/9*; for example, the initial tile arrangement in Fig. 2.45 will be represented by *state(3,4,5,8,0,2,7,1,6)*. (The zero stands for 'no tile'.) The *link/2* predicate is defined in *eight_links.pl* by focusing on the movement of the position with no tile; for example, two of the four states linked to the initial state in Fig. 2.45 are generated by means of the following clauses of *link/2*,

```
link(InState,OutState) :- down(InState,OutState).
link(InState,OutState) :- left(InState,OutState).
```

The pertinent clauses of *down/2* and *left/2* are respectively defined by

¹⁸This will be found by Breadth First or Iterative Deepening as these algorithms find a shortest route to the goal node. In cases where both these algorithms fail, the minimum number of moves to the goal state has been established by an appropriate informed search algorithm from Chap. 3.

```

?- consult(eight_puzzle).
% blindsearches compiled into blindsearches 0.00 sec, 7,408 bytes
% eight_links compiled into links 0.00 sec, 4,152 bytes
% eight_puzzle compiled 0.05 sec, 19,576 bytes
Yes
?- tiles.
Start state for test case number 1:
8 1 2
7 3
6 4 5
-----
...

-----
Start state for test case number 6:
3 4 5
8 2
7 1 6
-----
...

Select test case (a number between 1 and 11)... 6.
Select algorithm (df/df_dl/bf/bf_dl/bdf/id)... id.
% 2,299,419 inferences in 34.17 seconds (67294 Lips)
Solution in 16 steps.
Show result in full? (y/n) y.
3 4 5
8 2
7 1 6
-----
3 4 5
8 1 2
7 6
-----
3 4 5
8 1 2
7 6
-----
...

-----
1 3
8 2 4
7 6 5
-----
1 2 3
8 4
7 6 5
-----
Yes

```

Figure 2.46: Solving the Eight Puzzle

```
down(state(A,B,C,D,0,E,F,G,H),state(A,B,C,D,G,E,F,0,H)).  
left(state(A,B,C,D,0,E,F,G,H),state(A,B,C,0,D,E,F,G,H)).
```

Exercise 2.22. Complete the definition of *link/2*. ■

All the other problem relevant predicates are defined in the top module in `eight_puzzle.pl` which imports predicates from both `eight_links.pl` and `blindsearches.pl`.

Tail Recursion

If the last goal in the body of a recursive clause is the head, it is termed *tail recursive*. If all recursive clauses of a predicate are tail recursive, and a cut (!) precedes the last goal in each, the Prolog compiler will not retain reference to the earlier goals and the implementation will not crash due to stack overflow, and, it will run faster. Some compilers will recognize tail recursion automatically without the additional cut(s). It is good practice to use the cut for tail recursive code whatever system one uses.

The entries of Table 2.1 have been obtained by tail recursive versions using cuts. This is an important addition here as some test cases proved unsolvable without the additional cut.

Chapter 3

Informed Search

In this chapter we are going to discuss graph search algorithms and applications thereof for finding a *minimum cost* path from a start node to the goal node.

3.1 The Network Search Problem with Costs

The network search problem in Sect. 2.2 (Fig. 2.1) was devoid of any cost information. Let us now assume that the costs to traverse the edges of the graph in Fig. 2.1 are as indicated in Fig. 3.1.

There are two possible interpretations of the figures in Fig. 3.1: they can be thought of as costs of edge traversal or, alternatively, as edge lengths. (We prefer the latter interpretation in which case, of course, Fig. 3.1 is not to scale.) The task is to determine a minimum length path connecting s and g , or, more generally, minimum length paths connecting any two nodes.

The algorithms considered in this chapter assume the knowledge of an *heuristic distance* measure, H , between nodes. Values of H for the network in Fig. 3.1 are shown in Table 3.1. They are taken to be the estimated straight line distances between nodes and may be obtained by drawing the network in Fig. 3.1 to scale and taking measurements.

Three algorithms will be introduced here: the *A-Algorithm*, *Iterative Deepening A** and *Iterative Deepening A* $-\epsilon$* .

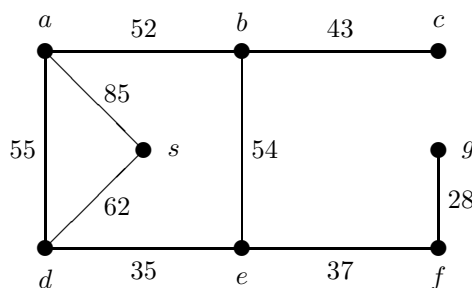


Figure 3.1: A Network with Costs

85	40	30	62	34	31	14	<i>s</i>
98	51	25	76	45	28	<i>g</i>	
109	71	54	73	37	<i>f</i>		
77	54	63	35	<i>e</i>			
55	61	88	<i>d</i>				
95	43	<i>c</i>					
52	<i>b</i>						
<i>a</i>							

Table 3.1: Straight Line Distances between Nodes in Fig. 3.1

3.1.1 Cost Measures

An estimated overall cost measure, calculated by the *heuristic evaluation function* F , will be attached to every path; it is represented as

$$F = G + H \quad (3.1)$$

where G is the *actual* cost incurred thus far by travelling from the start node to the current node and H , the

heuristic, is the *estimated* cost of getting from the current node to the goal node. Assume, for example, that in the network shown in Fig. 3.1 we start in d and want to end up in c . Equation (3.1) then reads for the path $d \rightarrow s \rightarrow a$ (with obvious notation) as follows

$$\begin{aligned} F(d \rightarrow s \rightarrow a, c) &= G(d \rightarrow s \rightarrow a) + H(a, c) \\ &= (62 + 85) + 95 = 147 + 95 = 242 \end{aligned} \quad (3.2)$$

3.1.2 The A-Algorithm

We know from Chap. 2 that for blind search algorithms the updating of the *agenda* is crucial: Breadth First comes about by appending the list of extended paths to the list of open paths; Depth First requires these lists to be concatenated the other way round.

For the A-Algorithm, the updating of the agenda is equally important. The new agenda is obtained from the old one in the steps ① and ② below.

- ① Extend the head of the old agenda to get a list of successor paths. An intermediate, ‘working’ list will be formed by appending the tail of the old agenda to this list.
- ② The new agenda is obtained by sorting the paths in the working list from ① in ascending order of their F -values.
- ③ The steps ① and ② are iterated until the path at the head of the agenda leads to the goal node.

In the example shown in Fig. 3.2, the paths are prefixed by their respective F -values and postfixed by their respective G -values. Using this notation and the cost information, the example path in (3.2) is now denoted by $242 - [a, s, d] - 147$. Notice that this path also features in Fig. 3.2.

It can be shown (e.g. [23]) that if the heuristic H is *admissible*, i.e. it never overestimates the actual minimum distance travelled between two nodes, the A-Algorithm will deliver a minimum cost path if such a path exists.¹In this case the A-Algorithm is referred to as an A^* -Algorithm and is termed *admissible*. (As the straight line distance is a minimum, the heuristic defined by Table 3.1 is admissible.)

Implementation

The predicate `a_search(+Start, +Goal, -PathFound)` in `asearches.pl` implements the A-Algorithm. A few salient features of `a_search/3` will be discussed only; for details, the reader is referred to the source code which broadly follows the pattern of implementation of the blind search algorithms (Fig. 2.15, p. 65 and Fig. 2.20, p. 69).

The implementation of the A-Algorithm in `asearches.pl` uses the built-in predicate `keysort/2` to implement step ② (see inset on p. 108).

The module invoking `a_search/3` should have defined (or imported) the following predicates.

- The connectivity predicate `link/2`. For the network search problem, this is imported from `links.pl` (Fig. 2.2, p. 49).
- The estimated cost defined by `e_cost/3`. For the network search problem, this is defined in `graph_a.pl` by

¹To be more precise, this holds only under some additional conditions which are satisfied, however, in most practical applications [23].

```
e_cost(Node,Goal,D) :- dist(Node,Goal,D).  
e_cost(Node,Goal,D) :- dist(Goal,Node,D).
```

with *dist/3* essentially implementing Table 3.1,

```
dist(s,a,85). ... dist(s,f,31). dist(s,g,14).  
dist(g,a,98). ... dist(g,f,28).  
...  
dist(b,a,52).
```

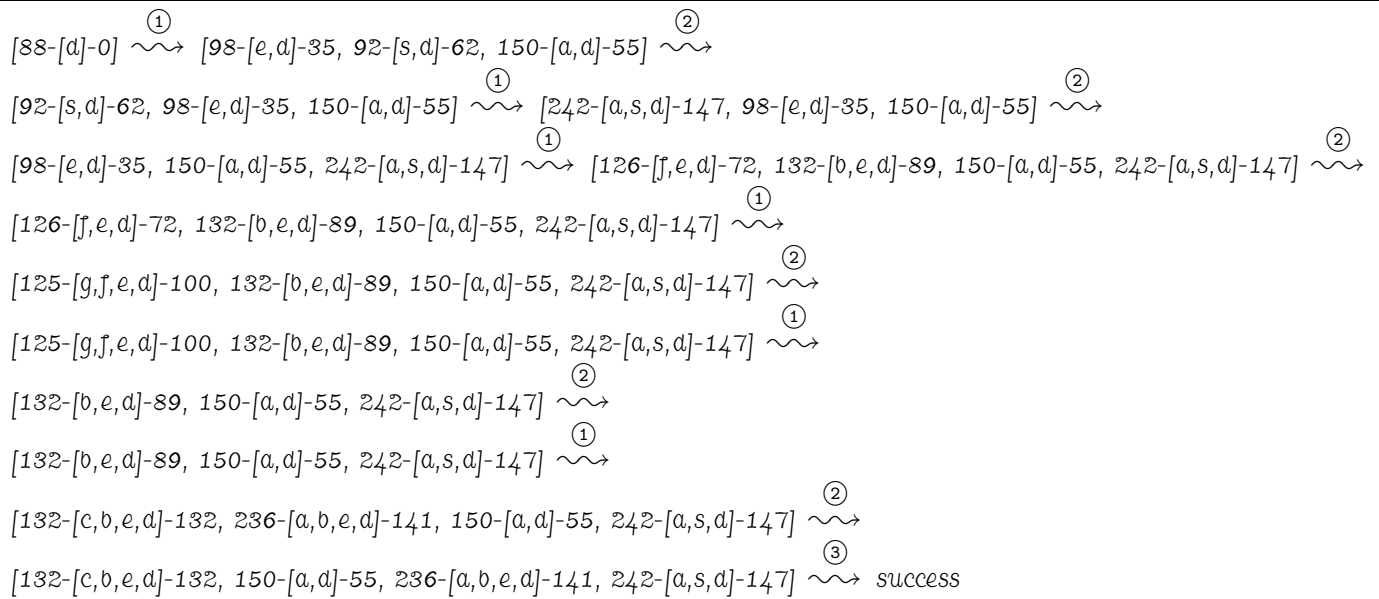


Figure 3.2: Hand Computations: The Evolution of the Agenda for the A-Algorithm (from d to c in Fig 3.1)

- The actual edge costs defined by *edge_cost/3*. For the network search problem, this is defined in *graph_a.pl* by

```
edge_cost(Node1,Node2,Cost) :- link(Node1,Node2),
                               e_cost(Node1,Node2,Cost).
```

Built-in Predicate: *keysort(+List,-Sorted)*

Unifies *Sorted* with the sorted version of *List*. The entries in *List* have to be in the form *key-term* and they will be sorted in ascending order of the value of *key*.

Example: Sort a list of names with ages according to increasing values of age. (Facts for *age/2* to be entered manually.)

```
?- consult(user).
|: age(adam,34).
|: age(tracy,18).
|: age(george,15).
|: [Ctrl]+[D]
% user compiled 0.00 sec, 480 bytes
Yes
?- bagof(_Age-_Name,age(_Name,_Age),L), keysort(L,Sorted).
L = [34-adam, 18-tracy, 15-george]
Sorted = [15-george, 18-tracy, 34-adam]
Yes
```

The interactive session below shows that the path $d \rightarrow e \rightarrow b \rightarrow c$ is a shortest one from d to c .

```
?- consult(graph_a).
% asearches compiled into a_ida_idaeps 0.00 sec, 7,736 bytes
% links compiled into edges 0.00 sec, 1,804 bytes
% graph_a compiled 0.00 sec, 16,584 bytes
?- a_search(d,c,PathFound), total_cost(PathFound,Cost).
PathFound = [d, e, b, c]
Cost = 132
```

3.1.3 Iterative Deepening A^* and its ϵ -Admissible Version

Application of the A -Algorithm to a more substantial example in Sect. 3.2 will reveal that the A -Algorithm may fail due to excessive memory requirements.² Clearly, there is scope for improvement.

In the mid 1980s, a new algorithm was conceived by Korf [20] combining the idea of Iterative Deepening (Sect. 2.6) with a heuristic evaluation function; the resulting algorithm is known as *Iterative Deepening A^** (IDA^*).³ The underlying idea is as follows.

- Use Depth First as the ‘core’ of the algorithm.

²We can see at this stage already that there is a special case of the A -Algorithm where lots of memory is required: the A -Algorithm specializes into Breadth First if unit edge costs and the zero heuristic are assumed.

³Noteworthy is also a more recent work by Korf [21] analysing IDA^* .

- Convert the core into a kind of Bounded Depth First Search with the bound (the horizon) now not being imposed on the length of the paths but on their F -values.
- Finally, imbed this ‘modified’ Bounded Depth First Search into a framework which repeatedly invokes it with a sequence of increasing bounds. The corresponding sequence of bounds in Iterative Deepening was defined as a sequence of multiples of some constant increment; a unit increment in the model implementation. The approach here is more sophisticated. Now, in any given phase of the iteration, the next value of the bound is obtained as the minimum of the F -values of all those paths which had to be ignored in the present phase. This approach ensures that in the new iteration cycle the least number of paths is extended.

The pseudocode of IDA^* won’t be given here; it should be possible to reconstruct it from the above informal description. It can be shown that IDA^* is admissible under the same assumptions as A^* .

The so-called ϵ -admissible version of IDA^* ($IDA^*-\epsilon$) is a generalization of IDA^* . It is obtained by extending the F -horizon to

$$\epsilon + \text{the minimum of all } F\text{-values of paths ignored}$$

with some fixed $\epsilon \geq 0$. (It clearly specializes to IDA^* for $\epsilon = 0$.) This algorithm may ‘catch’ a solution which otherwise would fall just outside the current F -horizon. $IDA^*-\epsilon$ may therefore find suboptimal solutions with

```

?- consult(graph_a).
% asearches compiled into aida_ideaeps 0.00 sec, 7,736 bytes
% links compiled into edges 0.00 sec, 1,804 bytes
% graph_a compiled 0.00 sec, 16,584 bytes
Yes
?- path.
Select start node s, a, b, ..., f, g: d.
Select goal node s, a, b, ..., f, g: c.
Select algorithm (a/ida/ideaeps)... a.
% 586 inferences in 0.00 seconds (Infinite Lips)
Solution in 3 steps.
d -> e -> b -> c
Total cost: 132
Yes

```

Figure 3.3: An Interactive Session. (See Exercise 3.1.)

Node	1	2	3	4	5	6	7	8	9	10
Co-ordinates	(1, 4)	(2, 7)	(2, 9)	(3, 4)	(3, 5)	(3, 9)	(4, 1)	(4, 5)	(4, 9)	(5, 4)

Table 3.2: Node Co-ordinates in the Network in Fig. 3.4

broadly the same effort and memory as IDA^* .⁴

Both versions, IDA^* and $IDA^*-\epsilon$, are implemented in `asearches.pl`.

Exercise 3.1. Complete the definition of `graph_a.pl` to solve the network search problem in Fig. 3.1 as illustrated by the interactive session in Fig. 3.3. (The user should be able to run any of the three algorithms discussed here.) ■

Exercise 3.2. Fig. 3.4 shows a small directed network with the nodes' co-ordinates shown in Table 3.2. Let the length of an edge be the *city block* (or *Manhattan*) distance of its endpoints.⁵

- Find the shortest route from node 1 to node 10 *manually* by using the A -Algorithm with the straight line heuristic.
- Write a module (`graph_b.pl`, say), which uses `asearches.pl`, for finding the shortest route as before but now the user should be able to select the algorithm in the style shown in Fig. 3.3. ■

Exercise 3.3. (*Adjacency matrix*) To represent the network in Fig. 3.4, you will have directly defined the connectivity predicate `link/2` by a collection of facts.⁶ A more flexible and elegant alternative to record the connectivity of a network is by using an *adjacency matrix*. The entries of this are zero everywhere except for

⁴ $IDA^*-\epsilon$ may not return an optimal solution. An example for this will be seen in Sect. 3.2.

⁵The city block distance between two points is the shortest distance when measured in a zigzag parallel to the co-ordinate axes. Thus, for example, the nodes 6 and 8 are $|3 - 4| + |9 - 5| = 5$ units apart.

⁶In all likelihood the same goes for the predicate that you will have used to record the nodes' co-ordinates from Table 3.2.

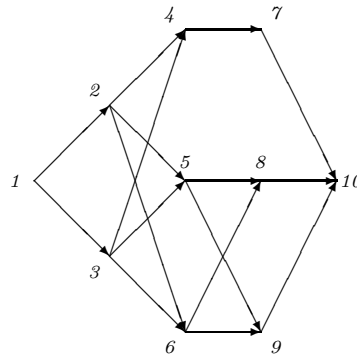


Figure 3.4: A Directed Network. (See Exercise 3.2.)

	1	2	3	4	5	6	7	8	9	10	
1	0	1	1	0	0	0	0	0	0	0	1
2	0	0	0	1	1	1	0	0	0	0	2
3	0	0	0	1	1	1	0	0	0	0	3
4	0	0	0	0	0	0	1	0	0	0	4
5	0	0	0	0	0	0	0	1	1	0	5
6	0	0	0	0	0	0	0	1	1	0	6
7	0	0	0	0	0	0	0	0	0	1	7
8	0	0	0	0	0	0	0	0	0	1	8
9	0	0	0	0	0	0	0	0	0	1	9
10	0	0	0	0	0	0	0	0	0	0	10

Figure 3.5: Adjacency matrix of the network in Fig. 3.4

positions (i, j) where there is a directed edge from node i to node j ; these entries are unity. Fig. 3.5 shows the adjacency matrix for the network in Fig. 3.4. Let this be defined by a Prolog fact such as

$$\begin{aligned} \text{adj}(1, & [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0], \\ & [0, 0, 0, 1, 1, 1, 0, 0, 0, 0], \\ & \dots \dots \dots \dots \dots \dots \dots \dots \dots \dots \\ & [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]). \end{aligned} \quad (3.3)$$

Let us also assume that the co-ordinates of the nodes from Table 3.2 are implemented by the Prolog fact

$$\text{co_ord}(1, [(1, 4), (2, 7), (2, 9), (3, 4), (3, 5), (3, 9), (4, 1), (4, 5), (4, 9), (5, 4)]).$$

- Define a predicate *make_links(+A)* which will write to the database the facts for *link/2* corresponding to the adjacency matrix *A*. Also define a predicate *make_co_ordinates(+C)* which takes a list of co-ordinates (list of pairs) *C* and writes to the database the corresponding facts in the form *in(Node, X_co_ord, Y_co_ord)*. (Remove old definitions from the database before writing to it.)
- Now, after revising your solution of Exercise 3.2, it should be possible to search the network in Fig. 3.4 thus

```
?- adj(1,A), co_ord(1,Co), path(A,Co).
Select start node 1, ..., 10: 1.
Select goal node 1, ..., 10: 10.
Select algorithm (a/ida/idaeps)... a.
% 561 inferences in 0.00 seconds (Infinite Lips)
Solution in 4 steps.
1 -> 2 -> 5 -> 8 -> 10
Total cost: 10
Yes
```

Notice in particular that the predicate *path(+A,+Co)* should initiate the search for the network with adjacency matrix *A* and list of node co-ordinates *Co*. Make use of *make_links/1* and *make_co_ordinates/1* from part (a) when defining *path/2*. Your implementation will be able to cope with any directed network specified in this manner. (Minor point: Display the correct number of nodes for the user to choose from.)

- (c) Use your implementation to determine the shortest path from node 1 to node 26 in the network in Fig. 3.6, p. 113. The node co-ordinates are given in Table 3.3, and, as before, the edge lengths should be calculated

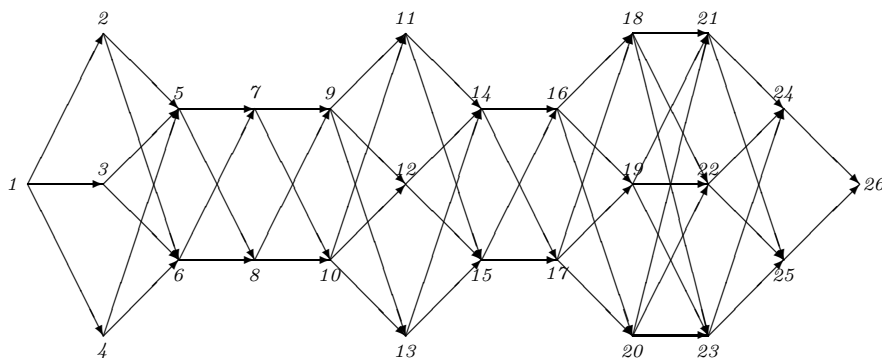


Figure 3.6: Network for Exercise 3.3, Part (c)

Node	1	2	3	4	5	6	7	8	9
Co-ordinates	(1, 2)	(2, 7)	(2, 14)	(2, 20)	(3, 2)	(3, 17)	(4, 5)	(4, 8)	(5, 2)

Node	10	11	12	13	14	15	16	17	18
Co-ordinates	(5, 20)	(6, 13)	(6, 17)	(6, 19)	(7, 2)	(7, 15)	(8, 7)	(8, 19)	(9, 4)

Node	19	20	21	22	23	24	25	26
Co-ordinates	(9, 8)	(9, 18)	(10, 3)	(10, 16)	(10, 19)	(11, 3)	(11, 12)	(12, 5)

Table 3.3: Node Co-ordinates in the Network in Fig. 3.6

by the city block distance.⁷

(The model solution for this exercise is in `graph_c.pl`.)

Exercise 3.4. (Sparsity) If the adjacency matrix of a network is *sparse*, i.e. most of its entries are zero (Fig. 3.5), it is a good idea to apply a compression scheme for storing it in the database. The following is a simple compression scheme. As each row can be thought of as a concatenation of lists comprising zeros and ones, we shall denote repetitions of the same character C by $N-C$ where N is the number of times the character C appears. Thus, for example, `[1-0, 2-1, 7-0]` will stand for the first row of the matrix in (3.3). Define a predicate `decompress(+C, -A)` for converting a compressed matrix C into the corresponding adjacency matrix A .⁸

Hint. A concise definition may be achieved by adopting the *functional programming style*:

1. Define a predicate for converting terms of the form $N-C$ to a list comprising N copies of C .
2. Define now a predicate by mapping the predicate in (1) followed by applying `flatten/2`.

⁷We shall meet this network in a different context in Sect. 3.4 as the search graph of the maze problem in Fig. 3.10, p. 122.

⁸The query in Exercise 3.3, part (b), may then equivalently be issued by

`?- c_adj(1, C), decompress(C, A), co_ord(1, Co), path(A, Co).`

if `c_adj/2` is used in an obvious manner for defining compressed adjacency matrices.

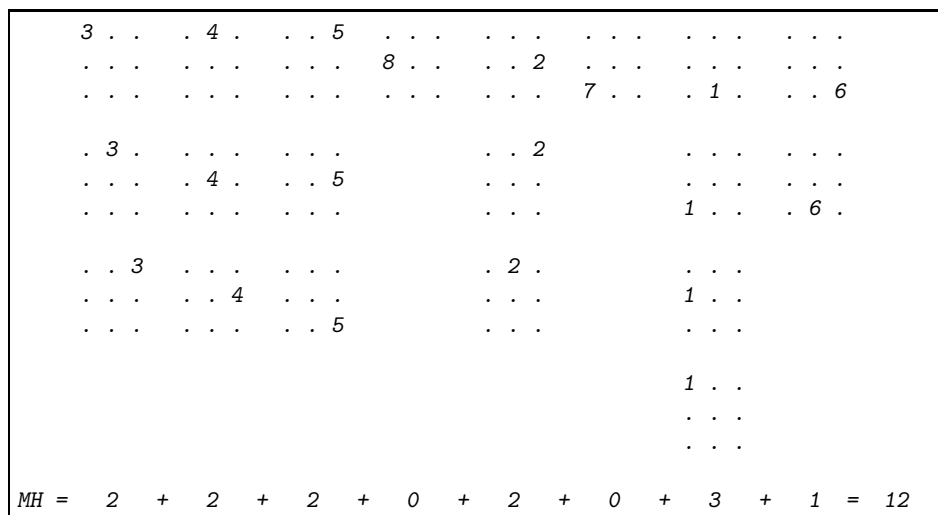


Figure 3.7: Calculating the Manhattan Distance between the tile arrangements in Fig. 2.45

3. Finally, implement decompression by mapping the predicate in (2) to the compressed matrix.

(The solution is in `graph_c.pl`.)

3.2 Case Study: The Eight Puzzle Revisited

For some choices of the terminal states for the Eight Puzzle we have not been able to find a solution using blind search (Table 2.1, p. 100). We are going to re-examine this puzzle here by informed search.

3.2.1 The Heuristics

A popular heuristic for the Eight Puzzle is the *Manhattan Distance* (MH). For two tile arrangements, the MH is the minimum total number moves all eight tiles need to be moved *individually* from their initial to their respective final positions. Whereas in the original version of the puzzle prior to moving a tile we had to make space by moving tiles which were ‘in the way’, now in this *relaxed* problem the obstacle tiles are simply ignored. (As before, moves sideways and up and down are allowed only.) For example, the MH between the tile arrangements in Fig. 2.45, p. 99, is 12 as shown in Fig. 3.7. The MH never exceeds the actual distance (i.e. the minimum number of moves needed to convey one configuration to the other) which is 16 here (Fig. 2.46, p. 101). The MH is therefore an admissible heuristic.

The predicate $\text{e_cost}(mh, +\text{State1}, +\text{State2}, -C)$ ⁹ returns the estimated cost between *State1* and *State2* as measured by the MH; for the states in Fig. 2.45 we have, for example,

```
?- e_cost(mh,state(3,4,5,8,0,2,7,1,6),state(1,2,3,8,0,4,7,6,5),C).
C = 12
```

⁹In the first argument we indicate the heuristic employed. (In Exercise 3.5 we will be considering another heuristic too.)

To implement this predicate, we first represent the system's states in matrix form, i.e. by a list comprising three lists.

```
matrix_form(state(T11,T12,T13,T21,T22,T23,T31,T32,T33),
             [[T11,T12,T13],[T21,T22,T23],[T31,T32,T33]]).
```

Given now two matrix representations, *Matrix1* and *Matrix2*, we find the number of steps *D* needed to convey the tile located at (i, j) in *Matrix1* to its new position in *Matrix2* by applying *mh_distance/5*, defined by

```
mh(I,J,Matrix1,Matrix2,D) :- ijth(I,J,Matrix1,E),
                             (E \= 0,
                              ijth(K,L,Matrix2,E),
                              D is abs(I - K) + abs(J - L));
                             D = 0), !.10
```

For example, the number of steps in the seventh sequence of tile moves in Fig. 3.7 is verified by

```
?- mh(3,2,[[3,4,5],[8,0,2],[7,1,6]],[[1,2,3],[8,0,4],[7,6,5]],D).
D = 3
```

Finally, as seen in Fig. 3.7, the MH between any two tile arrangements (in matrix notation) is the sum of the number of moves for each individual tile.

```
mh(Matrix1,Matrix2,D) :- mh(1,1,Matrix1,Matrix2,D11),
                           ...
                           mh(3,3,Matrix1,Matrix2,D33),
                           D is D11 + D12 + ... + D33.
```

Exercise 3.5. Another heuristic for the eight puzzle is the *number of misplaced tiles* (MP): each tile already in the right position will contribute zero whereas each of the other tiles will contribute unity. Implement this heuristic by *e_cost(mp,+State1,+State2,-C)*. Example:

```
?- e_cost(mp,state(3,4,5,8,0,2,7,1,6),state(1,2,3,8,0,4,7,6,5),C).
C = 6
```

Thus, this heuristic does not exceed the MH¹¹ which itself is admissible. Hence MP is admissible. (MP is defined in *eight_puzzle.a.pl*.) ■

3.2.2 Prolog Implementation

The Prolog implementation is in the file *eight_puzzle.a.pl*. A sample run is shown in Fig. 3.8, p. 116. For example, case 9 is now solvable while previously it was not viable (Table 2.1, p. 100). Table 3.4 shows the CPU times for the heuristic searches using a 300 MHz machine. (Unsuccessful cases and those with excessive computing times have been omitted.) Comparing Table 3.4 with Table 2.1 shows the dramatic benefit of using

¹⁰The predicate *ijth(?I,?J,+Matrix,?Entry)* is defined here by

```
ijth(I,J,ListOfRows,E) :- nth1(I,ListOfRows,Row), nth1(J,Row,E).
```

It is used in two modes. First, to get access to the (i, j) th entry of a matrix, use the mode *ijth(+I,+J,+Matrix,-Entry)*. Then, to identify the position of *Entry* in *Matrix*, use *ijth(-I,-J,+Matrix,+Entry)*.

¹¹In fact, MP is at most the number of tiles, i.e. 8. Since MH is 12 here, we know without checking further that MP is less than MH.

```

?- consult(eight_puzzle_a).
% asearches compiled into aida_ideaeps 0.00 sec, 7,704 bytes
% eight_links compiled into links 0.00 sec, 4,100 bytes
% eight_puzzle_a compiled 0.00 sec, 22,288 bytes
Yes
?- tiles.
Start state for test case number 1:
8 1 2
7 3
6 4 5
-----
...

-----
Start state for test case number 9:
5 6 7
4 8
3 2 1
-----
...

Select test case (a number between 1 and 10)... 9.
Select heuristic (mh/mp)... mh.
Select algorithm (a/ida/ideaeps)... a.
Solution in 30 steps.
Show result in full? (y/n) y.
5 6 7
4 8
3 2 1
-----
5 6 7
4 2 8
3 1
-----
...

-----
1 3
8 2 4
7 6 5
-----
1 2 3
8 4
7 6 5
-----
Yes

```

Figure 3.8: Solving the Eight Puzzle by Heuristic Search

<i>Test Case Number</i>			1	2	3	4	5	6	7	8	9	10
<i>Goal Node at Depth</i>			8	8	10	12	13	16	16	20	30	30
<i>CPU Seconds</i>	<i>mp</i>	<i>a</i>	0.1	0.1	0.0	0.3	0.7	26.8	14.3	-	-	-
		<i>ida</i>	0.1	0.1	0.1	0.5	1.0	4.2	5.1	59.9	-	-
	<i>mh</i>	<i>a</i>	0.0	0.0	0.1	0.1	0.1	0.9	0.7	38.0	42.0	-
		<i>ida</i>	0.1	0.1	0.0	0.1	0.1	0.3	0.8	8.1	2.8	52.9

Table 3.4: CPU Times (in Seconds) for the Eight Puzzle with Heuristic Search

heuristic search. It confirms furthermore that MH is better than MP and that IDA^* is preferable to the A^* -Algorithm.

Case 9 becomes viable for the number of misplaced tiles heuristic for $IDA^*-\epsilon$. With $\epsilon = 25$, we get a solution in 32 steps in 30.4 CPU seconds.

Exercise 3.6. (*Other Algorithms*) As precursors to the A -Algorithm, in many AI books two other algorithms are also discussed: *Hill Climbing* and *Best First Search* (e.g. [34]).

Hill Climbing is a modification of Depth First in that the nodes obtained by expanding a parent node will be, prior to them being put to the front of the agenda, sorted in ascending order of their estimated distances to the goal node.¹²

Best First is an extension of the previous idea in that now, prior to choosing the node to be expanded next, *all* open paths in the agenda are sorted in ascending order of their estimated distances to the goal node.¹³

You should implement these two algorithms.

Notes.

- (a) Model your implementation of the search algorithms on `asearches.pl`. As in `asearches.pl`, represent the estimated cost of a path by a prefix; no postfix is needed now.
- (b) Model your solution of the Eight Puzzle on `eight_puzzle_a.pl`.
- (c) Run the implementation and interpret the results.

(The model solution will be found in `bsearches.pl` and `eight_puzzle_b.pl`.) ■

3.3 Project: Robot Navigation¹⁴

Develop a Prolog program that can be used to guide a robot in the matrix shown in Fig. 3.9 along a *shortest* route from any cell to any other cell.¹⁵ The robot should be able to move parallel to the walls but not diagonally. *Notes.*

1. Use the search algorithms' implementations in `asearches.pl`.
2. Use the city block distance as a heuristic H .
3. There are several possibilities to model the 'cost' of a path. The simplest is to take its length as a measure of cost, i.e.

$$G = \text{path length} \quad (3.4)$$

The length is the sum of the edge costs each of which is in our application unity; we therefore declare

```
edge_cost(_,_,1).
```

Using this measure, the cost of the path found in Fig. 3.9 is 14.

4. Experiments using the cost measure in (3.4) suggest that the problem cannot always be solved by the A-Algorithm as the agenda may become excessively large. This will happen if there are too many paths of the same length sharing the same endpoints. The cost defined by

$$G = \text{path length} + \delta \times \text{path tortuosity} \quad (3.5)$$

¹²The underlying intuitive expectation is here that expanding nodes that are deemed closer to the goal node will lead faster to the goal node.

¹³Best First is therefore a kind of A-Algorithm with the G component in (3.1) set to zero.

¹⁴A simplified version of the problem described in this section served as a coursework problem in the late Prof. Imad Torsun's Prolog lectures in the late 1990s.

¹⁵The matrix layout (robot floorplan on Fig. 3.9) is taken from [23, p. 83].

```

?- consult(robot).
% rsearches compiled into rsearches 0.00 sec, 7,924 bytes
% floorplan compiled into floorplan 0.05 sec, 9,524 bytes
% robot compiled 0.05 sec, 25,116 bytes
Yes
?- robot.
      1  2  3  4  5  6  7  8  9 10 11 12 13 14
      .  .  .  .  .  .  .  .  .  .  .  .  .  .
+-----+
1 | | | | | | | | | | | | | | | | | 1
+-----+
2 | | | | | | | | | | | | | | | | | 2
+-----+
3 | | | | | | | | | | | | | | | | | 3
+-----+
4 | | | | | | | | | | | | | | | | | 4
+-----+
5 | | | XXX|XXX|XXX|XXX|XXX|XXX| | | | | | 5
+-----+
6 | | | XXX|XXX| | | XXX|XXX| | | | | | 6
+-----+
7 | | | XXX|XXX| | | XXX|XXX| | | | | | 7
+-----+
8 | | | | | | | | | | | | | | | | | 8
+-----+
9 | | | | | | | | | | | | | | | | | 9
+-----+
10 | | | | | | | | | | | | | | | | | 10
+-----+
11 | | | | | | | | | | | | | | | | | 11
+-----+
      1  2  3  4  5  6  7  8  9 10 11 12 13 14

Select start cell ... cell(5,11).
Select goal cell ... cell(7,3).
Select algorithm (a/ida/idaeps)... a.

% 842,633 inferences in 5.66 seconds (148875 Lips)
From cell(5, 11) to cell(7, 3) in 14 moves:
      1  2  3  4  5  6  7  8  9 10 11 12 13 14
      .  .  .  .  .  .  .  .  .  .  .  .  .  .
+-----+
1 | | | | | | | | | | | | | | | | | 1
+-----+
2 | | | | | | | | | | | | | | | | | 2
+-----+
3 | | | | | | | | * * * * * | | | | | | 3
+-----+
4 | | | | | | | | | | * | | | | | | | 4
+-----+
5 | | | XXX|XXX|XXX|XXX|XXX|XXX| * | | | | | 5
+-----+
6 | | | XXX|XXX| | | XXX|XXX| * | | | | | 6
+-----+
7 | | | XXX|XXX| | | XXX|XXX| * | | | | | 7
+-----+
8 | | | | | | | * * * * * * * * * | | | 8
+-----+
9 | | | | | | * | | | | | | | | | | | 9
+-----+
10 | | | | | * | | | | | | | | | | | 10
+-----+
11 | | | | | * | | | | | | | | | | | 11
+-----+
      1  2  3  4  5  6  7  8  9 10 11 12 13 14

Yes

```

Figure 3.9: Robot Navigation

where

$$\text{path tortuosity} = \text{number of turns}$$

will differentiate between such paths sufficiently enough for excessive growth of the agenda to be avoided. To guarantee that all least cost paths are also shortest paths, choose $\delta > 0$ small enough such that a shorter path, however tortuous, will always be assigned a smaller cost. Assuming that no path will have more than, say, nine turns, $\delta = 0.1$ will do. Using this measure, the cost of the path found in Fig. 3.9 is 14.3.¹⁶

5. Your implementation using (3.4) will always succeed if Iterative Deepening A^* is used but may run out of memory for the A -Algorithm.
6. A more ambitious implementation will use (3.5), and this will always succeed, also for the A -Algorithm. The implementations in `asearches.pl` can cope with the usual cost structure only, i.e. where each edge is assigned a fixed cost. To cater for the more complex cost structure in (3.5), you should devise a modified version of `asearches.pl`. (The model solution uses `rsearches.pl` that is an adaptation of `asearches.pl`.)
7. The predicate defining the floor layout, called `cell/2` in the model implementation, may be defined by facts as follows.

```
cell(1,1). cell(1,2). ... cell(11,11).
```

It would be rather tedious, however, to enter these facts into the database manually and therefore they are *asserted* ([9, p. 80]) by invoking a rule-based equivalent, `position/2`, prior to running the main body of the program. For example, the upper block of cell positions may be defined by

```
position(X,Y) :- between(1,11,X), between(1,4,Y).
```

which then is followed by the *assertion* of the facts defining `cell/2` by `layout/0` as shown below.

```
layout :- retractall(cell(_,_)),
           position(X,Y),
           assert(cell(X,Y)),
           fail.
layout.                                     } failure driven loop ([9, p. 77])
                                           } catch-all clause
```

This is a simple form of *memoization* (e.g. [19], p. 179 and [28], p. 181), aimed at saving computing time during the search process. In addition, it introduces some flexibility, as the suggested arrangement allows the floor layout to be easily modified if required.

8. The top level module of the model implementation is in `robot.pl`. It uses the modules in `rsearches.pl` (or `asearches.pl`, depending on which cost measure is being employed) and `floorplan.pl`. The latter implements the path's display on the terminal as shown in Fig. 3.9. (A less ambitious solution will display the path by showing its co-ordinates only.)

¹⁶By contrast, the path from `cell(5,11)` to `cell(7,3)` and having turns at `cell(5,8)`, `cell(9,8)`, `cell(9,4)` and `cell(7,4)` has the same length as the one found in Fig. 3.9 but it is more tortuous as it changes directions four times rather than thrice. It will be assigned the cost of 14.4.

3.4 Project: The Shortest Route in a Maze

Develop a Prolog program for searching for a shortest path in a maze of a specific kind with the following features.

- The program should search in mazes exemplified in Fig. 3.10 whereby
 - The gates are arranged in groups parallel to each other;
 - Adjacent groups of gates are a unit distance apart;
 - Groups of gates are numbered $1, 2, \dots$ (up to 12 in Fig. 3.10);
 - Group number 1 comprises the **IN** gate only;
 - The group with the highest number (here: 12) comprises the **OUT** gate only;
 - Gates are of unit width;
 - The position of the gates relative to the left wall is recorded by a number ($1, \dots, 20$ in Fig. 3.10) and the overall width of the maze is determined by the position of the rightmost gate;
- The program should display on the terminal the maze and the shortest path found.

Furthermore, as seen in Fig. 3.10, the program should also have the following features.

```

?- consult(maze).
% maze_disp compiled into display 0.05 sec, 18,816 bytes
% asearches compiled into a_ida_idaeps 0.00 sec, 7,660 bytes
% maze compiled 0.11 sec, 41,972 bytes
Yes
?- maze.
Select test case (a number between 1 and 5)... 2.
Select heuristic (zero/ed/alt)... ed.
Select algorithm (a/ida/idaeps)... a.

% 77,949 inferences in 0.55 seconds (141725 Lips)

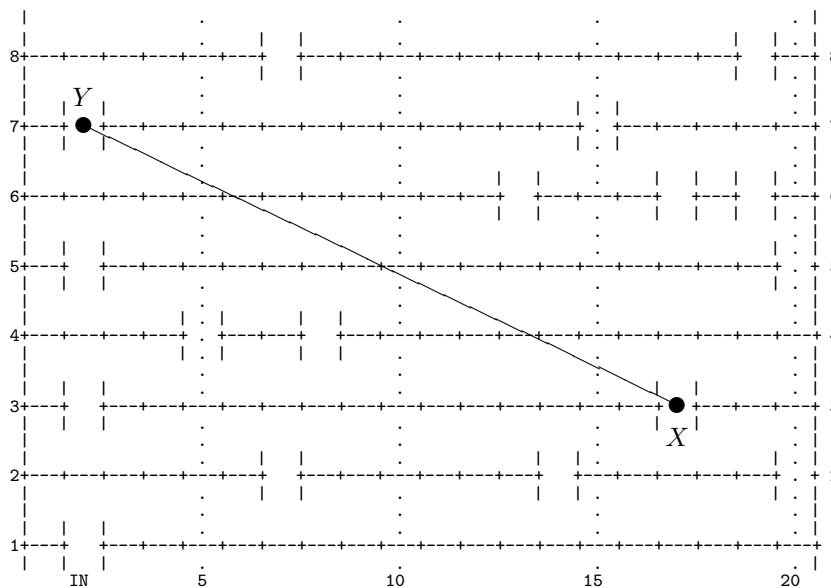
```

```

Length of shortest path is 54
Yes

```

Figure 3.10: Maze Search

Figure 3.11: Calculating the Euclidean Heuristic H_1

- The user should choose between three evaluation functions (of the form $F = G + H$), whereby the heuristic component, H , is one of the following: zero (**zero**), the Euclidean distance (**ed**), or, an alternative distance (**alt**) which will be described in Sect. 3.4.1. (All three suggested choices of H will be seen admissible.)
- The user should choose between three algorithms: A^* , Iterative Deepening A^* and Iterative Deepening $A^* - \epsilon$.
- The program should return a display of the shortest path found and its length.

3.4.1 Suggested Implementation Details

The predicate `gates/2` will be used to specify the structure of a maze. For example,

```
gates(2, [[2], [7,14,20], [2,17], [5,8], [2,20], [13,17,19],
          [2,15], [7,19], [4,8,18], [3,16,19], [3,12], [5]]).
```

specifies the maze shown in Fig. 3.10. The first argument of `gates/2` stands for the ‘test case number’; its second argument takes a list-of-lists defining the structure of the maze in an obvious manner.

Heuristics

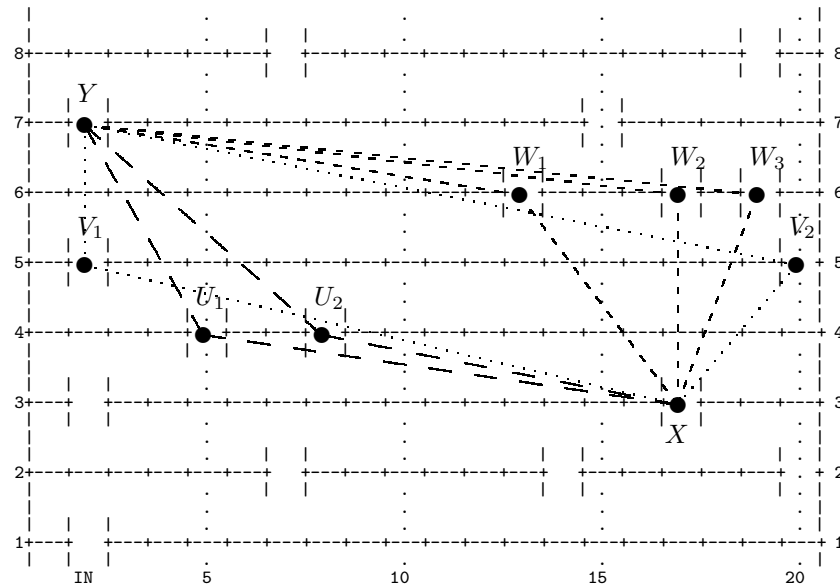
The zero heuristic H_0 . Put simply $H_0 \equiv 0$.

The Euclidean heuristic H_1 . This is the straight line (‘Euclidean’) distance e between any two gates. Fig. 3.11 illustrates H_1 : to estimate the distance between two gates X and Y , simply use Pythagoras (3.6).

$$H_1(X, Y) = e(X, Y) = \sqrt{(17 - 2)^2 + (3 - 7)^2} = 15.52 \quad (3.6)$$

The alternative heuristic H_2 . If X and Y are in adjacent rows then put $H_2(X, Y) = e(X, Y)$. Assume now that X and Y are at least two rows apart. $H_2(X, Y)$ is then defined with reference to Fig. 3.12. Take for each row of gates between X and Y every gate in that row as an intermediate gate in a two-stage ‘flight’ between X and Y . Keep the row fixed and compute the minimum of such ‘flight distances’ — each such minimum ‘flight distance’ is obviously a lower bound on the true maze distance between X and Y . The alternative heuristic $H_2(X, Y)$ is defined as the maximum of all such minimum flight distances, obtained by varying the in-between rows of gates. Equations (3.7)-(3.8) illustrate the computation of H_2 .

$$\begin{aligned}
 H_2(X, Y) = \max \{ & \min \{ e(X, U_1) + e(U_1, Y), \\
 & e(X, U_2) + e(U_2, Y) \}, \\
 & \min \{ e(X, V_1) + e(V_1, Y), \\
 & e(X, V_2) + e(V_2, Y) \}, \\
 & \min \{ e(X, W_1) + e(W_1, Y), \\
 & e(X, W_2) + e(W_2, Y), \\
 & e(X, W_3) + e(W_3, Y) \} \}
 \end{aligned} \tag{3.7}$$

Figure 3.12: Calculating the Alternative Heuristic H_2

$$H_2(X, Y) = \max \left\{ \begin{array}{l} \min \{ 16.28, 15.77 \}, \\ \min \{ 17.13, 21.72 \}, \\ \min \{ 16.05, 18.03, 20.62 \} \end{array} \right\} = 17.13 \quad (3.8)$$

The result is an admissible heuristic. Equations (3.6) and (3.7)-(3.8) show that H_2 is not worse than the Euclidean heuristic H_1 , i.e.

$$H_1(X, Y) \leq H_2(X, Y) \leq \text{true distance between } X \text{ and } Y$$

H_2 will be, however, more expensive to compute than either H_0 or H_1 .

Manual Implementation

As a first step towards a full implementation, the problem shall be solved for the maze in Fig. 3.10 with the zero heuristic H_0 and without returning a pictorial display of the path found. In this initial phase we won't be making use of *gates/2* directly. Instead, the necessary information about the maze will be represented by a collection of facts defining *edge_cost/3* thus

```
edge_cost(state(1,2),state(2,7),6).
edge_cost(state(1,2),state(2,14),13).
...
```

(The filename chosen to hold these clauses, *tedious.pl*, reflects the effort involved.) The above definition of *edge_cost/3* can be derived from the search graph indicated in Fig. 3.13 below. We define *link/2* in terms of *edge_cost/3* by

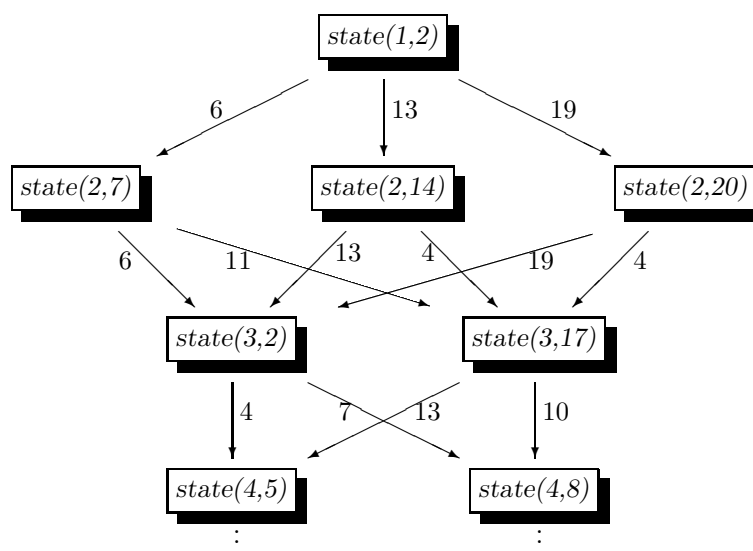


Figure 3.13: Search Graph for the Gates' Position

```
link(Node1,Node2) :- edge_cost(Node1,Node2,_).
```

The positions of the terminal gates will be recorded in `tedious.pl` by

```
start_state(state(1,2)). final_state(state(12,5)).
```

Finally, the zero heuristic will be implemented by the definition

```
e_cost(_,_,0).
```

We are now in a position to find interactively the path shown in Fig. 3.10:

```
?- consult(tedious).
% asearches compiled into aida_idaeps 0.00 sec, 7,704 bytes
% tedious compiled 0.00 sec, 15,544 bytes
Yes
?- start_state(_S), final_state(_G), a_search(_S,_G,_PathFound), write_term(_PathFound, []).
[state(1, 2), state(2, 7), state(3, 2), state(4, 5),
 state(5, 2), state(6, 13), state(7, 15), state(8, 7),
 state(9, 4), state(10, 3), state(11, 3), state(12, 5)]
Yes
```

Exercise 3.7. Complete the file `tedious.pl` and run the search for the maze in Fig. 3.10 by using the heuristic H_0 . ■

Full Implementation

The predicates which will be used by the search algorithms in `asearches.pl` should be defined in the top module, `maze.pl`, say. Below you will find some *guidelines* for these and another predicate used to display the result.

A *rule-based* version (in one clause) of `link/2` will define the node connectivity; then, for example, for the maze shown in Fig. 3.10 we get

```
?- consult(maze).
...
?- maze.17
Select test case (a number between 1 and 5)... 2.
Select heuristic (zero/ed/alt)... ed.
Select algorithm (a/ida/idaeps)... a.
...
?- link(state(3,17),Gate).
Gate = state(4, 5) ;
Gate = state(4, 8) ;
No
```

¹⁷This predicate, among other things, writes to the database the gates' arrangement chosen by the user. The predicate `gates/1` will be used to hold this information.

```
maze :- (retractall(gates(_));true),
        select_testcase(N),
        assert((gates(AllGates) :- gates(N,AllGates))),
        ...
```

Now you should define `link/2` for extracting the connectivity information from `gates/1`.

The predicate `e_cost(+Heur, +G1, +G2, -Est)` should return in `Est` the estimated distance of the gates `G1` and `G2`. Equations (3.6) and (3.7)-(3.8) are confirmed for example by

```
?- e_cost(ed, state(3,17), state(7,2), Est).
Est = 15.5242
?- e_cost(alt, state(3,17), state(7,2), Est).
Est = 17.1327
```

The pictorial display of the maze and the path found is accomplished by the predicate `show_picture(+Pic)`, defined in the module `maze_disp.pl`, with `Pic` specifying the maze and the path. To produce for example the display in Fig. 3.10, `Pic` will be unified with the list of pairs

```
[(5, [5]), (3, [3,12]), (3, [3,16,19]), ..., (2, [2])]
```

(`Pic` allows to identify for each row the gate through which the path passes and the position of all the gates in that row.)

Exercise 3.8. Complete the implementation of the maze search problem as described above. ■

Exercise 3.9. The model implementation uses the straight line distance to derive heuristics. Modify the implementation by basing the heuristics on the city block distance and observe and interpret changes in the CPU time. ■

Exercise 3.10. The idea of the alternative heuristic function H_2 can be refined. For example, $H_3(X, Y)$ may be defined for gates X and Y at least three rows apart by maximizing the minimum flight distances between X and Y with two intermediate gates. Put $H_3(X, Y) = H_2(X, Y)$ if X and Y are less than three rows apart. H_n ($n \geq 4$) may be defined in an analogous manner. H_n is a better heuristic than H_{n-1} , i.e. $H_n \geq H_{n-1}$ but it will be more expensive to compute. Experiment with these heuristics to find out whether the computational benefit in the search process outweighs the increased computing time for the heuristics themselves. ■

Exercise 3.11. The search graph of the maze problem is *acyclic*, i.e. no node can be visited more than once (e.g. Fig. 3.13). Path checking is therefore not required in this case. Disable path checking in `asearches.pl` and confirm that the resulting implementation uses less CPU time. ■

3.5 Project: Moving a Knight

Write a Prolog program which, given two positions on the chessboard, will find a shortest sequence of moves a knight needs between these two positions.¹⁸ Your program will behave as indicated in Fig. 3.14. You should experiment with the suggested heuristics to find out how long the search takes with each.

The model solution is in `knight.pl` and it uses `asearches.pl`.

¹⁸The present search problem originates from [10].

```

?- consult(knight).
% asearches compiled into a_ida_idaeps 0.00 sec, 7,704 bytes
% knight compiled 0.05 sec, 19,104 bytes
Yes
?- jumps.
Select heuristic (min/mh/ed/co)... ed.
Select algorithm (a/ida)... ida.
Select initial position of knight ([a-h][1-8])... a8.
Select final position of knight ([a-h][1-8])... h1.
cost limit/CPU time: 1/399.3
cost limit/CPU time: 4.42719/399.35
cost limit/CPU time: 4.49285/399.35
cost limit/CPU time: 4.52982/399.35
cost limit/CPU time: 4.60768/399.35
cost limit/CPU time: 4.61245/399.41
cost limit/CPU time: 4.63246/399.46
cost limit/CPU time: 4.84391/399.52
cost limit/CPU time: 4.89443/399.63
cost limit/CPU time: 5.2249/399.74
cost limit/CPU time: 5.23607/399.9
cost limit/CPU time: 5.26491/400.06
cost limit/CPU time: 5.40588/400.23
cost limit/CPU time: 5.40832/400.39
cost limit/CPU time: 5.41421/400.61
cost limit/CPU time: 5.44721/400.94
cost limit/CPU time: 5.72029/401.33
cost limit/CPU time: 5.78885/401.77
cost limit/CPU time: 5.84708/402.26
cost limit/CPU time: 5.86356/402.76
cost limit/CPU time: 5.89737/403.31
cost limit/CPU time: 6/403.91
% 474,024 inferences in 4.66 seconds (101722 Lips)
Solution in 6 steps:
a8 b6 a4 b2 d1 f2 h1

  +---+---+---+---+---+---+---+---+
8 | X |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
6 |   | X |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
5 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
4 | X |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
3 |   |   |   |   |   |   |   |   |
  +---+---+---+---+---+---+---+---+
2 |   | X |   |   |   | X |   |   |
  +---+---+---+---+---+---+---+---+
1 |   |   |   | X |   |   |   | X |
  +---+---+---+---+---+---+---+---+
    a   b   c   d   e   f   g   h
Yes

```

Figure 3.14: Sample Session: Moving a Knight

Suggested Heuristics

Let the letters annotating the board's columns be replaced by $1, \dots, 8$ and refer to the knight's position by a pair $P = (x, y)$ with co-ordinates $x, y \in \{1, \dots, 8\}$. Define two heuristics H_1 and H_2 by

$$H_q(P, P') = \begin{cases} \frac{d_1(P, P')}{3}, & \text{when } q = 1 \\ \frac{d_2(P, P')}{\sqrt{5}}, & \text{when } q = 2 \end{cases} \quad (3.9)$$

where d_1 and d_2 denote respectively the city block distance (also called 'Manhattan distance') and the Euclidean distance:

$$d_q((x, y), (x', y')) = \begin{cases} |x - x'| + |y - y'|, & \text{when } q = 1 \\ \sqrt{(x - x')^2 + (y - y')^2}, & \text{when } q = 2 \end{cases}$$

H_1 and H_2 are referred to in Fig. 3.14 by *mh* and *ed*, respectively.

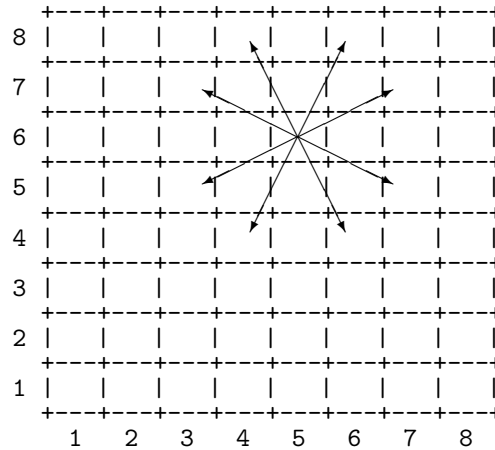


Figure 3.15: The Knight Moves One Step

An interesting property of these heuristics is that none dominates the other.¹⁹

Admissibility. We show that both H_1 and H_2 are admissible. For pairs of positions one step apart, it is

$$d_q(P, P') = \begin{cases} 3, & \text{when } q = 1 \\ \sqrt{5}, & \text{when } q = 2 \end{cases}$$

(This is illustrated in Fig. 3.15 for $P = (4, 6)$.) In general, if the sequence of positions

$$P = P_0, P_1, \dots, P_n = P'$$

takes the knight from P to P' in the *minimum* number of moves n , say, then, by the *Triangle Inequality* for d_q it is

$$\begin{aligned} d_q(P, P') &= d_q(P_0, P_n) \\ &\leq d_q(P_0, P_1) + \dots + d_q(P_{n-1}, P_n) = \begin{cases} 3n, & \text{when } q = 1 \\ \sqrt{5}n, & \text{when } q = 2 \end{cases} \end{aligned} \quad (3.10)$$

From (3.10) we have by the definition of H_q in (3.9) that

$$H_q(P, P') \leq n$$

Generalization. We note in passing that for any $q \geq 1$, H_q , defined by

$$H_q(P, P') = \frac{d_q(P, P')}{(1 + 2^q)^{1/q}}$$

with

$$d_q((x, y), (x', y')) = (|x - x'|^q + |y - y'|^q)^{1/q}$$

¹⁹By this we mean that there are positions P, P', Q and Q' such $H_1(P, P') < H_2(P, P')$ and $H_1(Q, Q') > H_2(Q, Q')$. This holds for example for $P = (4, 3)$, $P' = (7, 4)$, $Q = (4, 3)$ and $Q' = (6, 1)$.

is an admissible heuristic.²⁰

Combined heuristic. This we define by

$$H_{co}(P, P') = \max\{H_1(P, P'), H_2(P, P')\}$$

It is of course also admissible and it is a genuine improvement on both H_1 and H_2 since, as we have seen earlier, none dominates the other.

A Non-Admissible Heuristic. Define H_{min} by

$$H_{min}((x, y), (x', y')) = \min\{|x - x'|, |y - y'|\}$$

This is not admissible since $H_{min}((7, 2), (1, 8)) = 6$ but $(7, 2) \rightarrow (5, 3) \rightarrow (3, 4) \rightarrow (2, 6) \rightarrow (1, 8)$ is a sequence of 4 moves from $(7, 2)$ to $(1, 8)$. IDA^* will indeed find this non-optimal sequence of moves if it is used with H_{min} .

²⁰The reasoning is as before with the following addenda. It is

$$d_q(P, P') = \|P - P'\|_q$$

with the q -norm $\|\cdot\|_q$ defined by

$$\|(x, y)\|_q = (|x|^q + |y|^q)^{1/q}$$

The Triangle Inequality for d_q follows from the *Minkowski Inequality* for the q -norm

$$\|P + P'\|_q \leq \|P\|_q + \|P'\|_q$$

See, e.g. [31].

Chapter 4

Text Processing

Whereas the problems considered thus far were taken from Artificial Intelligence, we are going now to apply Prolog to problems in text processing.

The present chapter is in three parts.

First, the Prolog implementation is described of a tool for removing from a file sections of text situated between marker strings. (The tool is therefore a primitive static *program slicer*; [32] and [12].) This tool then is used in a practical context for removing sample solutions from the L^AT_EX source code of a solved exam script. It is also shown in this context how SWI-Prolog code can be embedded into a LINUX shell script.

The second part addresses the question of how Prolog can be used to generate L^AT_EX code for drawing *parametric curves*. Some new features of Prolog will thereby also be introduced.

The final part comprises a sequence of solved Prolog exercises, implementing a tool for drawing *families* of parametric curves in L^AT_EX. The exercises are of increasing complexity and finally describe how SWI-Prolog can interact with LINUX through a shell script.

4.1 Text Removal

4.1.1 Practical Context

I use L^AT_EX on LINUX for preparing examination papers. This is done in the following steps.

1. Create a L^AT_EX source file in a text editor.
2. Translate the L^AT_EX file into a DVI file.
3. Translate the DVI file into a PDF file.
4. View the PDF file.

These steps are performed for `exam.tex` by running the LINUX commands in Fig. 4.1.¹ Upon execution of the last line in Fig. 4.1, a new window will pop up and the exam paper may be viewed.

External examiners require examination papers *with model answers*. I create therefore a PDF file with model solutions in the first instance where answers are appended to each subquestion. The answers are placed between

¹`bash-3.1$` is the system prompt in Fig. 4.1.

```
bash-3.1$ latex exam.tex
bash-3.1$ dvipdf exam.dvi
bash-3.1$ kpdf exam.pdf
```

Figure 4.1: Processing the File `exam.tex`

some *marker strings* enabling me eventually to *locate and remove* all text between them when creating the final L^AT_EX source leading to the printed PDF for students. It is this *text removal* process which is automated by the Prolog implementation to be discussed here.

4.1.2 Specification

Write a predicate *sieve(+Infile,-Outfile,+Startmarker,+Endmarker)* of arity 4 for removing all text in the file named in *Infile* in between all occurrences of lines starting with text in *Startmarker* and those starting with text in *Endmarker*. The result should be saved in the file named in *Outfile*. *Outfile* is without marker lines. If *Outfile* already exists, its old version should be overwritten, if it does not exist, it should be newly created. The file shown in Fig. 4.2 is an example of *Infile* with the marker phrases ‘water st’ and

'water e', say. (The file comprises a random collection of geographical names.) After the Prolog query

```

birmingham
new york
lake district

las vegas
grand canaria

london
water starts
pacific ocean
loch ness

north sea
water ends
kalahari desert
st andreas fault
north pole
water starts
mediterranean sea
lake balaton
lake konstanz
river thames
river danube
water ends
britain
europe

```

The diagram shows a list of geographical names. Four specific lines are highlighted with arrows and labels in rounded boxes:

- An arrow points to the line `water starts` with the label *Line starting with Startmarker*.
- An arrow points to the line `water ends` with the label *Line starting with Endmarker*.
- An arrow points to the line `water starts` with the label *Line starting with Startmarker*.
- An arrow points to the line `water ends` with the label *Line starting with Endmarker*.

Figure 4.2: The File `with_waters`

```
?- sieve('with\_waters', 'without\_waters', 'water st', 'water e').2
Yes
```

the file `without_waters` will have been created. This is shown in Fig. 4.3.

4.1.3 Implementation

Definition of Predicates

The main predicate `sieve/4` is defined in terms of `sieve/2`, both are shown in (P-4.1).

²Notice that the sequence of *two* characters '`\_`' represents the underscore. Likewise, '`\.`' will have to be typed for the dot in a filename or marker string.

```

birmingham
new york
lake district

las vegas
grand canaria

london
kalahari desert
st andreas fault
north pole
britain
europe

```

Figure 4.3: The File without_waters

Prolog Code P-4.1: Definition of sieve/4 and sieve/2

```

1 sieve(File_In, File_Out, Start_String, End_String) :-
2     see(File_In),
3     tell(File_Out),
4     told,
5     append(File_Out),
6     switch_off,
7     sieve(Start_String, End_String),
8     told,
9     seen, !.

10 sieve(Start_String, End_String) :-
11     atom_chars(Start_String, Start_List),
12     atom_chars(End_String, End_List),
13     get_line(Line),
14     ((append(Start_List,_,Line), switch_on); true),
15     (Line = [end_of_file];
16     atom_codes(A,Line),
17     ((switch(off), write(A)); true),
18     ((append(End_List,_,Line), switch_off); true),
19     sieve(Start_String, End_String)).

```

The predicates *get_line/1* (and its auxiliary *get_line/2*), *switch_off/1* and *switch_on/1* are defined in (P-4.2).

Prolog Code P-4.2: Auxiliaries for (P-4.1)

```
1 :- dynamic(switch/1).  
2 switch_off :- retractall(switch(_)),  
3             assert(switch(off)).  
4 switch_on  :- retractall(switch(_)),  
5             assert(switch(on)).  
6 get_line(List) :- get_line([], List).  
7 get_line(Acc, List) :- get_char(Next),  
8                       ((Next = '\n', reverse([Next|Acc], List));  
9                       (Next = end_of_file, List = [Next]));  
10                      get_line([Next|Acc], List)).
```

For the SWI-Prolog built-ins *atom_chars/2* and *atom_codes/2*, the reader is referred respectively to pages 126 and 19 of [9].

Noteworthy are three more built-in predicates used here: the standard Prolog predicates *see/1*, *seen/0* (respectively for directing the input stream to a file and redirecting it) and *get_char/1* for reading a character; the example below illustrates their use by reading the first three characters of the file *with_waters* in Fig. 4.2.

```
?- see(with_waters), get_char(First), get_char(Sec), get_char(Third), seen.
First = b
Sec = i
Third = r

Yes
```

Details of Implementation

- The predicate *get_line/1* in (P-4.2) is defined in terms of *get_line/2* by the accumulator technique. It reads into its argument the next line from the input stream. Example:

```
?- set_prolog_flag(toplevel_print_options, [max_depth(20)]).
Yes
?- see(with_waters), get_line(First), get_line(Sec), seen.
First = [b, i, r, m, i, n, g, h, a, m,
]
Sec = [n, e, w, , y, o, r, k,
]
Yes
```

The following observations apply.

1. It is seen from the above query that a line read by *get_line/1* is represented as a list of the characters it is composed of.
 2. By definition the last character of each line in a file is the new line character ‘\n’. That explains the line break seen in the above query.
 3. Finally (not demonstrated here), each file ends with the end-of-file marker ‘end_of_file’. The one-entry list [end_of_file] is deemed to be the last line of every file by the definition in (P-4.2).
- The switches *switch_off/0* and *switch_on/0* are used, writing respectively *switch(off)* and *switch(on)* in the Prolog database, respectively for removal and retention of lines from the input file.
 - The main predicates are *sieve/4* and *sieve/2* in (P-4.1), the latter defined by recursion and called by the former.
- sieve/4*: this is the top level predicate.

1. Line 2 opens the input file.
2. The goals in lines 3-4 in (P-4.1) make sure that the earlier version of the output file (if there is such a file) is deleted.
3. In line 5, the new output stream is opened via *append/1*³.
4. In line 6, the switch is set to the position (‘off’), anticipating that initially lines will be retained.

³Not to be confused with the predicate *append/3*!

5. In line 7, *sieve/2* is invoked and processing is carried out.
6. Lines 8 and 9 close respectively output and input.

sieve/2: this is called from *sieve/4*.

1. Lines 14 and 18 contain the most interesting feature of this predicate: *append/3* is used in them for *pattern matching*. For example, the goal

```
append(Start_List,_,Line)
```

succeeds if the initial segment of the list *Line* is *Start_List*.

2. *atom_chars/2* is used in *sieve/2* to disassemble the start and end markers into lists in preparation for pattern matching.
3. Notice that the built-in predicate *atom_codes/2* can be used in *two roles* as the interactive session below demonstrates.

```
?- atom_codes(A,[b, i, r, m, i, n, g, h, a, m]).
A = birmingham
Yes
?- atom_codes(birmingham, L).
L = [98, 105, 114, 109, 105, 110, 103, 104, 97, 109]
Yes
```

In line 16 of (P-4.1), *atom_codes/2* is used in its first role, i.e. to convert a list of characters to an atom. This atom is the current line, it is written to the output file.

4. Recursion is stopped in *sieve/2* (and control is returned to line 8 of *sieve/4*) when the end-of-file marker is read (line 15).

4.1.4 Using a LINUX Shell Script

Specification

Imbed the Prolog implementation from Sect. 4.1.3 into a LINUX shell script for providing the same functionality as the predicate *sieve/4* does. The application obtained thereby will run without explicitly having to use the SWI-Prolog system. The intended behaviour of the script is illustrated in Fig. 4.4. The dialogue shown in Fig. 4.4 has the same effect as the Prolog session envisaged in Sect. 4.1.2.

[22] is an accessible introduction to LINUX and the beginnings of shell scripting.

Implementation

Plan

```
bash-3.1$ ./sieve with\_waters without\_waters water\ st water\ e
% /home/acsenki/scripts/sieve.pl compiled 0.00 sec, 4,284 bytes
Input file : 'with\_waters'
Output file: 'without\_waters'
Text removal between the phrases 'water st' and 'water e'
bash-3.1$ cat without\_waters
bermingham
new york
lake district

las vegas
grand canaria

london
kalahari desert
st andreas fault
north pole
britain
europe
```

Figure 4.4: Running the Shell Script `sieve`

The shell script should

1. Receive four arguments from the user (two filenames and two pattern strings),
2. Write them to a temporary file `temp`,
3. Invoke SWI-Prolog in the batch mode, which then
 - Should open the temporary file `temp`,
 - Should read the strings from `temp`,
 - Should call `sieve/4` to perform text removal,
 - Should close `temp`
4. Close the Prolog system,
5. Report on the actions performed,
6. Delete `temp`.

Shell Script and Additional Prolog Predicates

The LINUX shell script `sieve` in (S-4.1) is an implementation of the plan.

LINUX *Shell Script S-4.1: sieve*

```

1  #!/bin/bash
2  if [ $# -ne 4 ]; then
3      echo "Error: supply four arguments"
4  else
5      if [ -e $1 ]; then
6          echo $1 > temp
7          echo $2 >> temp
8          echo $3 >> temp
9          echo $4 >> temp
10     #
11     pl -f sieve.pl -g go -t halt
12     #
13     echo "Input file : '$1'"
14     echo "Output file: '$2'"
15     echo "Text removal between the phrases '$3' and '$4'"
16     #
17     rm temp
18     else
19         echo "Error: file '$1' does not exist"
20     fi
21 fi

```

In line 11 of (S-4.1), the Prolog source `sieve.pl` is invoked as a command line argument [33, Sect. 2.3]. `sieve.pl` comprises (P-4.1), (P-4.2) from Sect. 4.1.3 and the code in (P-4.3).

Prolog Code P-4.3: Definition of *go/0* and *get_string/1*

```

1 go :- see(temp),
2     get_string(File_In),
3     get_string(File_Out),
4     get_string(Start_String),
5     get_string(End_String),
6     sieve(File_In, File_Out, Start_String, End_String),
7     seen.
8
9 %
10 % auxiliary predicate get_string/1 ...
11 %
12
13 get_string(String) :- get_line(List),
14                      append(ShortList, ['\n'],List),
15                      atom_chars(String, ShortList).

```

In *go/0* from *sieve.pl* the existence of a file named **temp** is assumed, comprising four lines, the two file names (input and output files) and the two marker patterns, forming one line each. The top level predicate is now *go/0* which then uses *sieve/4*.

Running the Script

The script **sieve** makes (and eventually deletes) a temporary file **temp**, holding the four strings read by the predicate *go/0*. The script invokes the Prolog source *sieve.pl*, effecting a result as specified in Sect. 4.1.2. Some additional features are also demonstrated in the LINUX command window Fig. 4.5.

```

bash-3.1$ chmod -x sieve
bash-3.1$ ls -l sieve
-rw--w----+ 1 acsenki 2042 426 Sep 2 16:11 sieve
bash-3.1$ ./sieve with\_waters without\_waters water\ st water\ e
bash: ./sieve: Permission denied
bash-3.1$ chmod +x sieve
bash-3.1$ ./sieve with\_waters without\_waters water\ st
Error: supply four arguments
bash-3.1$ ./sieve with\_waters without\_waters water\ st water\ e
Input file : 'with_waters'
Output file: 'without_waters'
Text removal between the phrases 'water st' and 'water e'
bash-3.1$ ls temp
ls: temp: No such file or directory

```

Figure 4.5: Another Run of the Shell Script **sieve***Comments on Fig. 4.5.*

1. The first three commands illustrate what happens if initially **sieve** is not executable.
2. The fourth command makes **sieve** executable.
3. The fifth command illustrates the script's response if less than four arguments are supplied.

4. The next command shows the normal mode of operation. The response has to be read in conjunction with (S-4.1). The output file created is `without_waters`; it is of course identical to that in Fig. 4.3.
5. The last command confirms that the temporary file `temp` has been removed.

4.1.5 Application: Removing Model Solutions

`part_sln.tex` (shown in Fig. 4.6) is a file forming part of a collection of $\text{\LaTeX}$ source files to be assembled to a single $\text{\LaTeX}$ source. Text between the user-defined $\text{\LaTeX}$ commands `\solstart` and `\solend` forms part of a

```

...
\definecolor{hellgrau}{gray}{0.85}
\newcommand{\solstart}{\begin{center}\textbf{- - - - -}}
    \fcolorbox{black}{hellgrau}{Start Solution}- - - - -\end{center}}
\newcommand{\solend}{\begin{center}\textbf{- - - - -}}
    \fcolorbox{black}{hellgrau}{End Solution}- - - - -\end{center}}
...
\begin{itemize}
\item
First question.
\item
Second question.
\end{itemize}
\solstart
\begin{itemize}
\item
Answer to first question.
\item
Answer to second question.
\end{itemize}
\solend
Further questions.
...

```

Figure 4.6: The File `part_sln.tex`

model solution of exam questions, not to be shown to students in the final version. Fig. 4.7 shows the structure of the printed version of the exam script *with* solutions.

The task is to use the shell script `sieve` for producing the file `part.tex` from `part_sln.tex`; the latter is

```
...  
  
    • First question.  
    • Second question.  
  
- - - - - Start Solution - - - - -  
  
    • Answer to first question.  
    • Answer to second question.  
  
- - - - - End Solution - - - - -  
  
Further questions.  
...
```

Figure 4.7: Structure of the Printed Exam Script with Solutions

shown in Fig. 4.8. In `part.tex`, all lines between `\solstart` and `\solend` have been removed, including the marker lines themselves.

```
...
\definecolor{hellgrau}{gray}{0.85}
\newcommand{\solstart}{\begin{center}\textbf{- - - - -}}
\fcolorbox{black}{hellgrau}{Start Solution}- - - - -\end{center}}
\newcommand{\solend}{\begin{center}\textbf{- - - - -}}
\fcolorbox{black}{hellgrau}{End Solution}- - - - -\end{center}}
...
\begin{itemize}
\item
First question.
\item
Second question.
\end{itemize}
Further questions.
...
```

Figure 4.8: The File `part.tex`

It is seen in Fig. 4.8 in particular that the text between the marker phrases (`\solstart` and `\solend`) is removed only if they are the first phrase of their respective lines. (This is why the command definitions in Fig. 4.8 are still there.)

```
bash-3.1$ ./sieve part\_sln\.tex part\.tex \solstart \solend
% /home/acsenki/scripts/sieve.pl compiled 0.01 sec, 4,284 bytes
Input file : 'part\_sln.tex'
Output file: 'part.tex'
Text removal between the phrases '\solstart' and '\solend'
```

Figure 4.9: Running the Shell Script `sieve`

The task was achieved by running the shell script as shown in Fig. 4.9. Fig. 4.9 illustrates how string arguments containing the backslash character or the dot are used when running the shell script.

4.2 Text Generation and Drawing with L^AT_EX

4.2.1 Cycloids

Cycloids are a class of plain curves, well known from the Calculus of Variations (see e.g. the early classic [13, p. 26] or [26, Ch. 22, p. 844]). A cycloid is described by a point P attached to a disc rolling on a straight line (the *base line*) (Fig. 4.10). The following notation will be used.

- r is the radius of the disc,
- a is the distance of $P = (x, y)$ from the disc's centre C ,
- ϕ is the angle of rotation of the disc, measured in radians, clockwise positive.

The disc rests initially on the co-ordinate origin, therefore, $C = (0, r)$ and $P = (0, r - a)$ for $\phi = 0$; this is the disc on the left in Fig. 4.10. If P is outside the disc ($a > r$) the curve generated is a *prolate* cycloid (Fig. 4.11); if it is inside ($a < r$) a *curtate* cycloid is obtained (Fig. 4.12); and, if it is on the perimeter of the disc ($a = r$) a *common* cycloid (Fig. 4.13) is obtained. (For cycloids and other plane curves a good reference is [11, p. 165].) The co-ordinates of a point on the cycloid are given by

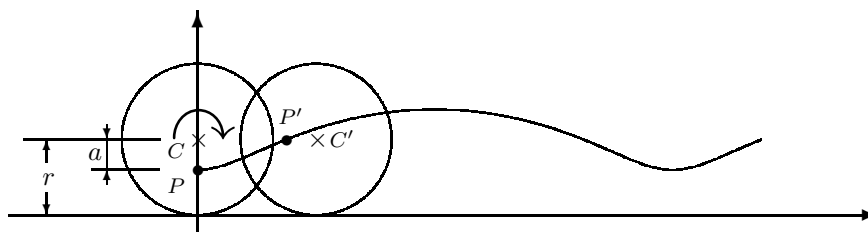


Figure 4.10: Drawing a Cycloid ($\phi = \pi/2$)

$$x = r\phi - a \sin \phi, \quad (4.1)$$

$$y = r - a \cos \phi. \quad (4.2)$$

The disc on the right in Fig. 4.10 is obtained by rotating the initial disc clockwise by $\phi = \pi/2$. According to (4.1)-(4.2), P 's new position is $P' = (r\phi - a \sin \phi, r - a \cos \phi) = (r\pi/2 - a, r)$, whereas C obviously moves to $C' = (r\phi, r) = (r\pi/2, r)$.

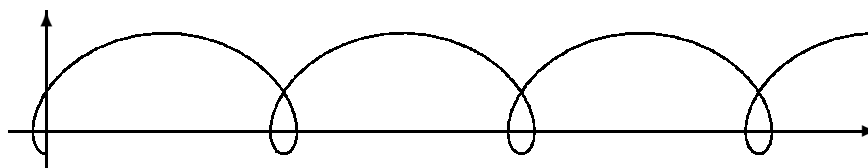


Figure 4.11: Prolate Cycloid Drawn with `\writecurve` from Fig. 4.14 ($r = 5$, $a = 8$, 3.5 revs)

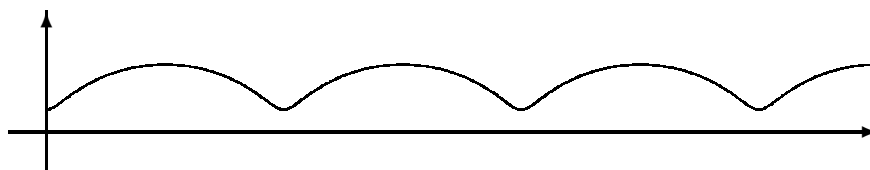


Figure 4.12: Curtate Cycloid Drawn with `\writecurve` similar to Fig. 4.14 ($r = 5$, $a = 3$, 3.5 revs)

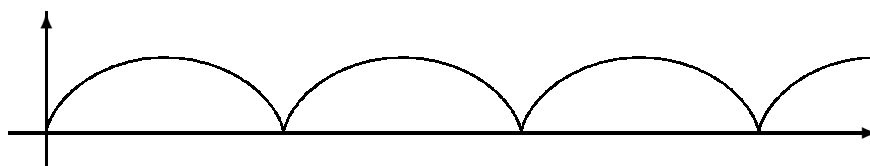


Figure 4.13: Common Cycloid Drawn with `\writecurve` similar to Fig. 4.14 ($r = 5$, $a = 5$, 3.5 revs)

4.2.2 Task

Define a Prolog predicate which will generate a `LTEX` command for drawing a cycloid of a given description.

The only tool available is the $\text{\LaTeX}$ package `epic` (e.g. [14]).

The package `epic` provides the command `\drawline` for connecting a sequence of points by a straight line segment. The syntax of this command is

`\drawline[stretch](x1,y1)(x2,y2)...(xn,yn)`

where *stretch* is an optional parameter (not used here) and $(x_1, y_1)(x_2, y_2) \dots (x_n, y_n)$ is the sequence of coordinates of the points to be connected. The task is to define a Prolog predicate *define_command/4* for displaying on the terminal text which is essentially the $\text{\LaTeX}$ command sought. This is illustrated in Fig. 4.14. The text so obtained is then pasted (after possibly some minor modifications) into the desired location in the

```
?- define_command(5, 8, 3.5, 100).
\newcommand{\writecurve}{\drawline(0,-3)(-0.645588,-2.80733)
(-1.20712,-2.23862)(-1.60458,-1.32124)(-1.76588,-0.099392)(-1.63027,1.36808)
...
(101.754,11.3212)(104.35,12.2386)(107.111,12.8073)(109.956,13.0)}
Yes
```

Figure 4.14: Generating the $\text{\LaTeX}$ Command `\writecurve` with *define_command/4*

$\text{\LaTeX}$ source file. The curve thus drawn will comprise a sequence of straight line segments, an approximation to the specified cycloid, looking like a smooth curve if the subdivision of the parameter interval is fine enough. Fig. 4.11, for example, was drawn by applying the $\text{\LaTeX}$ code (L-4.1). (The $\text{\LaTeX}$ command `\writecurve`, as generated by Prolog in Fig. 4.14, is used in line 9 of (L-4.1).)

$\text{\LaTeX}$ Code L-4.1: Drawing Fig. 4.11

```
1 \begin{figure}[h]
2 \begin{center}
3 \setlength{\unitlength}{1mm}
4 \begin{picture}(118,16)(0,0)
5 \thicklines
6 \put(5,-5){\vector(0,1){21}}
7 \put(0,0){\vector(1,0){115}}
8 \thinlines
9 \put(5,5){\makebox(0,0){\writecurve}}
10 \end{picture}
11 \end{center}
12 \caption{Prolate Cycloid Drawn with \texttt{\writecurve} from
13 Fig.~\ref{textprocessing:cycloids:generatecommand}
14 ($r=5$, $a=8$, $3.5$ revs)}
15 \label{textprocessing:cycloids:fig:prolate}
16 \end{figure}
```

4.2.3 Solution

The Prolog predicates for generating the $\text{\LaTeX}$ command `\writecurve` are shown in (P-4.4).

Prolog Code P-4.4: Prolog Code Generating \writecurve

```

1 cyc(R, A, Alpha, Pair) :- Pi is 3.1415926,
2                           Rad is Alpha * Pi / 180,
3                           S is sin(Rad),
4                           C is cos(Rad),
5                           X is R * Rad - A * S,
6                           Y is R - A * C,
7                           concat_atom(['(',X,',',Y,')'], Pair).

8 mesh(Revs, NInt, List) :- mesh(Revs, NInt, NInt, List, []), !.

9 mesh(_, _, 0, [0|Acc], Acc).
10 mesh(Revs, NInt, NumInt, List, Acc) :-
11     H is NumInt * (Revs * 360 / NInt),
12     NewNumInt is NumInt - 1,
13     mesh(Revs, NInt, NewNumInt, List, [H|Acc]).

14 pairs(R, A, Revs, NInt, Pairs) :- mesh(Revs, NInt, Mesh),
15                                   maplist(cyc(R,A), Mesh, Pairs).

16 define_command(R, A, Revs, NInt) :-
17     pairs(R, A, Revs, NInt, Pairs),
18     concat_atom(['\\newcommand{\\writecurve}{\\drawline'|Pairs], Atom),
19     concat_atom([Atom, '}', ''], C),
20     write(C).

```

Comments on, and Exemplification of (P-4.4).

- ① Let $r = 10$, $a = 4$ and $C = (0, 10)$. A counterclockwise rotation by $\alpha = 90^\circ$ (& associated roll of the disc to the right) moves the point $P = (0, 6)$ to $P' = (11.708, 10.0)$.

```

?- cyc(10, 4, 0, Pair).
Pair = '(0,6)'
Yes
- cyc(10, 4, 90, Pair).
Pair = '(11.708,10.0)'
Yes

```

`cyc/3` is essentially an implementation of (4.1)-(4.2) with the proviso that rotations are measured in degrees. The output of `cyc/3` is an atom.

- ② Let us assume that we want to plot the path of P between the two positions from ①, involving a quarter turn clockwise. A crude approximation will take snapshots corresponding to the positions 0° , 15° , 30° , 45° , 60° , 75° and 90° . The number of intervals involved is therefore 6 (each of length 15°). The 7 gridpoints are generated as a list by `mesh/3` thus

```

?- mesh(0.25, 6, List).
List = [0, 15, 30, 45, 60, 75, 90]
Yes

```

- ③ A sequence of points on the path of P is generated by *pairs/5*. For example, the 7 pairs of co-ordinates of P in ② are obtained by

```
?- pairs(10, 4, 0.25, 6, Pairs).
Pairs = ['(0,6)', '(1.58272,6.1363)', '(3.23599,6.5359)', '(5.02555,7.17157)',
        '(7.00787,8.0)', '(9.22627,8.96472)', '(11.708,10.0)']
Yes
```

pairs/5 uses *mesh/3* as an auxiliary. Furthermore, *cyc/5* is used in *partial application* in the second goal in the definition of *pairs/5* in the first argument of *maplist/3*. The output of *pairs/5* is a list of atoms. They represent the co-ordinates of the points which will form the vertices of the approximating polygon. `\drawline` from *epic* will be used to connect them.

- ④ *define_command/4* essentially concatenates the list entries from ② thus

```
?- define_command(10, 4, 0.25, 6).
\newcommand{\writecurve}{\drawline(0,6)(1.58272,6.1363)(3.23599,6.5359)(5.02555,7.17157)
                             (7.00787,8.0)(9.22627,8.96472)(11.708,10.0)}
Yes
```

- ⑤ Numbers whose modulus is very small or very large are displayed by default in Prolog in the scientific number format (the ‘exponential notation’). **If applicable**, change such numbers to be displayed in the floating point format using the ‘non-exponential notation’. For example, $1/888888$ will be displayed as $1.125e-06$. Change this to 0.000001125 in the $\text{\LaTeX}$ file.⁴ (Notice that this point does not apply to the output generated in ④.)
- ⑥ Now the $\text{\LaTeX}$ command `\writecurve` is ready to be used inside a figure and it will draw the desired cycloid. Fig. 4.15 was drawn with the `\writecurve` $\text{\LaTeX}$ command from ④; the code for Fig. 4.15 is not shown here as it is very similar to that shown in (L-4.1).

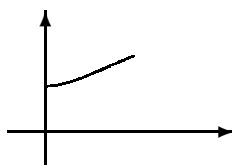


Figure 4.15: ‘Quarter’ Cycloid Drawn with `\writecurve` ($r = 10$, $a = 4$, $1/4$ revs)

4.3 Exercises

Exercise 4.1. The predicate `sieve/4` was defined in Sect. 4.1 for *removing* text situated between some specified pairs of markers. Define now a predicate `retain/4` for *retaining* text between some specified pairs of markers. (Such a predicate could be used, for example, for extracting all figures from a $\text{\LaTeX}$ document.) Use your Prolog implementation in a shell script for solving the same task. ■

Exercise 4.2. The two circles shown in Fig. 4.10 were drawn with the user-defined $\text{\LaTeX}$ command `\defcirc`. The definition of `\defcirc` was generated interactively by running the predicate `circ_command/4` as shown in Fig. 4.16. (L-4.2) shows a partial view of the $\text{\LaTeX}$ `picture` environment defining Fig. 4.10: lines

```
?- circ_command(10, 0, 0, 100).
\newcommand{\defcirc}{\drawline(10,0)(9.98027,0.627905)
(9.92115,1.25333)(9.82287,1.87381)(9.68583,2.4869)(9.51057,3.09017)
...
(9.82287,-1.87381)(9.92115,-1.25333)(9.98027,-0.627906)(10.0,-1.0718e-06)}
Yes
```

Figure 4.16: Generating the $\text{\LaTeX}$ Command `\defcirc` with `circ_command/4`

9 and 11 illustrate the use of `\defcirc`.

⁴The alternative is using `sformat/3` (formatted write) in (P-4.4) for displaying numbers in non-exponential notation; see Exercise 4.3.

AT_EX Code L-4.2: Partial view of the AT_EX code for Fig. 4.10

```

1 \begin{figure}[h]
2 \begin{center}
3 \setlength{\unitlength}{1mm}
4 \begin{picture}(118,25)(0,0)
5 \thicklines
6 \put(25,-5){\vector(0,1){30}}
7 \put(0,-2){\vector(1,0){115}}
8 \thinlines
9 \put(25,8){\makebox(0,0){\defcirc}}
10 ...
11 \put(40.707963,8){\makebox(0,0){\defcirc}}
12 ...
13 \end{picture}
14 \end{center}
15 \caption{Drawing a Cycloid}\label{textprocessing:fig:definingcycloid}
16 \end{figure}

```

Define the Prolog predicate *circ_command(+Radius, +CentreX, +CentreY, +NInt)*⁵ for displaying on the terminal AT_EX code defining `\defcirc`.

As before, assume that only *basic* AT_EX and the *epic* package are available.⁶ ■

Exercise 4.3. You will have defined in Exercise 4.2 a Prolog predicate *circ_command/4* the output of which may have to be put through the manual processing step described in ⑤ of Sect. 4.2.3. This exercise is about writing an *improved* implementation of *circ_command/4*, called *imp_circ_command/4*, that will obviate this since its output will contain pairs of numbers in non-exponential notation only.

The ‘old’ version of the predicate may be used to define a command for a circle of radius 10 with centre (0, 10) by approximating the circle with a regular 20 sided polygon (Fig. 4.17). Both entries of the sixteenth

```

?- circ_command(10, 0, 10, 20).
\newcommand{\defcirc}{\drawline(10,10)(9.51057,13.0902)
(8.09017,15.8779)(5.87785,18.0902)(3.09017,19.5106)(2.67949e-07,20.0)
(-3.09017,19.5106)(-5.87785,18.0902)(-8.09017,15.8779)(-9.51057,13.0902)
(-10.0,10.0)(-9.51057,6.90983)(-8.09017,4.12215)(-5.87785,1.90983)
(-3.09017,0.489435)(-8.03847e-07,3.19744e-14)(3.09017,0.489435)(5.87785,1.90983)
(8.09017,4.12215)(9.51056,6.90983)(10.0,10.0)}
Yes

```

Figure 4.17: Generating the AT_EX Command `\defcirc` with *circ_command/4*

pair in Fig. 4.17 are in the exponential notation, something AT_EX won’t accept. The modified version produces essentially the same output with all the numbers in the floating point notation (Fig. 4.18).

You should define *imp_circ_command/4* by using the SWI-Prolog built-in predicate *sformat/3*.

Hint.

The predicate *sformat/3* is there for producing formatted output returned as a string. Use the ‘f’ format (for floating point, non-exponential) in the second argument of *sformat/3*. For further information, see [6, p. 493]

⁵*NInt* denotes the number of intervals used when discretising a full revolution.

⁶In basic AT_EX `\circle` is used to draw circles. It allows, however, to draw circles up to a certain size only.

```
?- imp_circ_command(10, 0, 10, 20).
\newcommand{\defcirc}{\drawline(10.0000000,10.0000000)
(9.5105652,13.0901699)(8.0901700,15.8778524)(5.8778527,18.0901698)
(3.0901701,19.5105651)(0.0000003,20.0000000)(-3.0901696,19.5105653)
(-5.8778522,18.0901702)(-8.0901697,15.8778529)(-9.5105650,13.0901704)
(-10.0000000,10.0000005)(-9.5105653,6.9098306)(-8.0901703,4.1221480)
(-5.8778531,1.9098305)(-3.0901707,0.4894351)(-0.0000008,0.0000000)
(3.0901691,0.4894346)(5.8778518,1.9098295)(8.0901694,4.1221467)
(9.5105648,6.9098291)(10.0000000,9.9999989)}
Yes
```

Figure 4.18: Generating the L^AT_EX Command `\defcirc` with `imp_circ_command/4`

and [33].

Exercise 4.4. We are now in a position to address the generation of L^AT_EX code for *any* parametric two-dimensional curve. The aim is to define a predicate

$$\textit{gen_command2}(+CName, +Fun, +Lower, +Upper, +NInt, +Pars) \quad (4.3)$$

The arguments and the intended working of `gen_command2/6` are best explained with reference to an example.

The curve we are going to use is the improved circle *imp_circ*/5 from (P-A.11), p. 193 (solution of Exercise 4.3).

The L^AT_EX command for drawing a polygonal approximation with four sides to the lower half of a circular arc with radius 10, centre (0, 10) should be generated thus

```
?- gen_command2('\halfcirc', imp_circ, 180, 360, 4, [10,0,10]).
\newcommand{\halfcirc}{\drawline(-10.0000000,10.0000005)(-7.0710683,2.9289327)(-0.0000008,0.0000000)
(7.0710671,2.9289315)(10.0000000,9.9999989)}
```

Once this command definition is in the L^AT_EX code, `\halfcirc` is ready to be used in a figure. (The output may then look like the polygon in Fig. 4.19.) The arguments in (4.3) are easily matched to their respective values

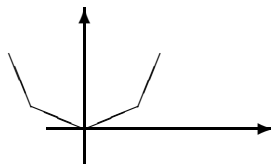


Figure 4.19: Polygon Drawn with `\halfcirc`

in the query. On the other hand, *imp_circ*(+R, +X, +Y, +Alpha, -Pair), the predicate from (P-A.11), has

1. Three fixed (input) parameters: radius *R*, and the two co-ordinates of the centre *X* and *Y*;
2. One argument: angle of rotation *Alpha*, measured counterclockwise positive from the circle's rightmost point;
3. One output: *Pair*, returned as a string.

The following is taking place in the query above.

- The command name *CName* in (4.3) is unified with the string '`\halfcirc`';
- The predicate name *Fun* is unified with '`imp_circ`';
- The domain of the argument *Alpha* is the interval $[Lower, Upper] = [180, 360]$. It is subdivided into *NInt* (= 4) intervals of equal length. The function values (pairs) are calculated internally for all interval endpoints, i.e. the 5 values of *Alpha*, [180, 225, 270, 315, 360];
- The argument *Pars* (list of parameters) is unified with [10, 0, 10], amounting to the unifications $R = 10$, $X = 0$, $Y = 10$;
- And, finally, after some processing, the command definition is written to the terminal.

Built-in Predicate: *apply(+Pred, +List)*

Uses the entries of *List* as arguments to the predicate *Pred*. Partial application of *Pred* is possible. The examples below refer to a polynomial defined by the predicate *pol/5*,

*pol(A, B, C, X, Y) :- Y is A + B * X + C * X^2.*

```
?- pol(4, 3, 2, 10, Y).
Y = 234
Yes
?- apply(pol, [4, 3, 2, 10, Y]).
Y = 234
Yes
?- apply(pol(4, 3), [2, 10, Y]).
Y = 234
Yes
```

apply/2 is a *higher order* predicate. Use *apply(+Pred, +List)* to invoke *Pred* whose arity is not known at compile time.

Detailed Plan.

The main point is to recognize the need to be able to pass on a predicate name as an argument. The built-in predicate *apply/2* is used to accomplish that. The implementation described here has a ‘functional flavour’.

1. Write a predicate *gen_mesh(+Lower, +Upper, +NInt, -Mesh)* for generating a list of meshpoints.

```
?- gen_mesh(180, 360, 4, Mesh).
Mesh = [180, 225, 270, 315, 360]
Yes
```

2. Define a predicate *applic(+Fun, +Pars, +Argument, -Outcome)* for calculating values of a function, defined by a predicate. For example, instead of having

```
?- imp_circ(10, 0, 10, 225, Outcome).
Outcome = '(-7.0710683,2.9289327)'
Yes
```

we may now equivalently do

```
?- applic(imp_circ, [10, 0, 10], 225, Outcome).
Outcome = '(-7.0710683,2.9289327)'
Yes
```

The two queries may deliver the same but the second one will be preferable in our context as it allows the predicate name to be passed on as an *argument*; *applic/4* is therefore a *higher order* predicate. Notice that the order of the arguments supplied to *Fun* is replicated by the entries of the list *Pars* and the arguments *Argument* and *Outcome*.

Hint. Use the built-in predicate *apply/2*. (See inset.)

3. Define a predicate *gen_vals(+Fun, +Lower, +Upper, +NInt, +Pars, -Vals)* for calculating the list of values taken by a given function at equidistant gridpoints. Example:

```
?- gen_vals(imp_circ, 180, 360, 4, [10,0,10], Vals).
Vals = ['(-10.0000000,10.0000005)', '(-7.0710683,2.9289327)', '(-0.0000008,0.0000000)',
        '(7.0710671,2.9289315)', '(10.0000000,9.9999989)']
Yes
```

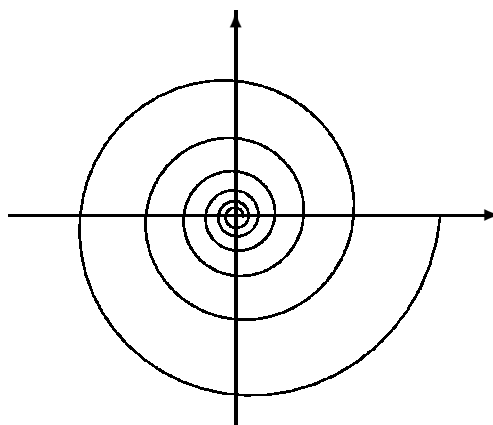
Use here *gen_mesh/4* and *applic/4* from above. Furthermore, use also the built-in predicate *maplist/3*.

4. Finally define *gen_command2(+CName, +Fun, +Lower, +Upper, +NInt, +Pars)*; it should behave as exemplified on p. 154.

■

Exercise 4.5. The *logarithmic spiral* in Fig. 4.20 was drawn with the L^AT_EX command `\spiral` the definition of which was generated with Prolog by using *gen_command2/6* from Exercise 4.4.

```
?- gen_command2('\spiral', log_spiral, 0, 2160, 300, [85, 0, 0]).
\newcommand{\spiral}{\drawline(1.0000000,0.0000000)(1.0030823,0.1267188)(0.9901165,0.2542187)
...
(25.6446869,-6.5844539)(26.5581065,-3.3550864)(27.0651201,-0.0000174)}
Yes
```

Figure 4.20: Logarithmic Spiral Drawn with `\spiral`

Define the predicate `log_spiral(+Alpha, +CentreX, +CentreY, +RotAngle, -Pair)` and then redraw in $\text{\LaTeX}$ the spiral on Fig. 4.20.

Hint. As is well known (e.g. [2]), a point on the logarithmic spiral with Cartesian co-ordinates $(r \cos \phi, r \sin \phi)$ is defined by $r = e^{k\phi}$ with $k = \cot \alpha$, where (r, ϕ) are the point's polar co-ordinates and α is the constant (acute) angle at which the spiral cuts all rays emitted from the origin. (ϕ and α are both measured in radians in these formulae.) In the above query, we have made $2160^\circ/360^\circ = 6$ revolutions, subdivided the interval $[0^\circ, 2160^\circ]$ into 300 intervals of equal length, and, the angle α measured 85° . (Obviously, the arguments *Alpha* and *RotAngle* in `log_spiral/5` are both measured in degrees.) The pole was taken to be the origin $(0, 0)$.

Note. An entire section is devoted to spirals in the beautiful book [25]. Questions concerning their self-similarity occupy the authors' attention. ■

Exercise 4.6. You are asked to defined the predicate `curves/2` in this exercise. It will simplify and automate the command definitions considered in Exercise 4.4.

Assume that we want to draw possibly *several* parametric curves in $\text{\LaTeX}$ each of which we can in isolation specify, generate and draw as described in Exercise 4.4. The pasting-in from the terminal of the $\text{\LaTeX}$ codes generated is cumbersome and error prone as it is a manual step. Therefore, we want to be able to create a file where all the $\text{\LaTeX}$ code will be deposited, ready to be included into our $\text{\LaTeX}$ document via `\include`. Furthermore, the curves' *interactive* specifications (via the keyboard) is also best avoided for the same reason; the preferred way of doing this is via some input file.

Illustrative Example.

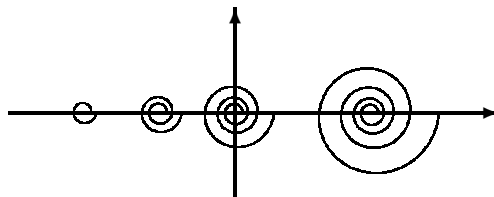


Figure 4.21: Growing Spirals

We want to generate Fig. 4.21 containing four spirals. The L^AT_EX command for each of the four spirals can be generated by *gen_command2/6* from Exercise 4.4. (It is assumed of course that the predicate *log_spiral/5* from Exercise 4.5 is available.) Once *curves/2* is available, we can solve this task in the following three steps.

- ① Create a file stating the four curves' specifications in terms of *gen_command2/6*; this has been done here in *spirals* shown in Fig. 4.22. The lines in *spirals* whose first character is % serve as comment lines.

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Spirals specified via gen_command2/6 ...
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% gen_command2('\tinyspiral', log_spiral, 0, 360, 36, [85, 0, 0]). ...
%
gen_command2('\tinyspiral', log_spiral, 0, 360, 36, [85, 0, 0])
%
% gen_command2('\smallspiral', log_spiral, 0, 720, 72, [85, 0, 0]). ...
%
gen_command2('\smallspiral', log_spiral, 0, 720, 72, [85, 0, 0])
%
% gen_command2('\normalspiral', log_spiral, 0, 1080, 108, [85, 0, 0]). ...
%
gen_command2('\normalspiral', log_spiral, 0, 1080, 108, [85, 0, 0])
%
% gen_command2('\largespiral', log_spiral, 0, 1440, 144, [85, 0, 0]). ...
%
gen_command2('\largespiral', log_spiral, 0, 1440, 144, [85, 0, 0])
%

```

Figure 4.22: The File *spirals*

- ② Perform now the following Prolog dialogue.

```

?- consult(draw).
% draw compiled 0.00 sec, 11,432 bytes
Yes
?- curves('spirals', 'spirals.tex').
Yes

```

- ③ The file `spirals.tex` will have been created in step ②. This is shown in Fig. 4.23. Notice that

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Spirals specified via gen_command2/6 ...
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% gen_command2('\tinyspiral', log_spiral, 0, 360, 36, [85, 0, 0]). ...
%
\newcommand{\tinyspiral}{\drawline(1.0000000,0.0000000)(0.9999608,0.1763201)
...
(1.6805635,-0.2963289)(1.7327464,-0.0000002)}
%
...
%
% gen_command2('\largespiral', log_spiral, 0, 1440, 144, [85, 0, 0]). ...
%
\newcommand{\largespiral}{\drawline(1.0000000,0.0000000)(0.9999608,0.1763201)
...
(8.7429878,-1.5416285)(9.0144653,-0.0000039)}
%

```

Figure 4.23: The File `spirals.tex`

`spirals.tex` is a valid L^AT_EX file best included into the L^AT_EX source by means of `\include{spirals}`. Lines starting in `spirals` with `%` are copied unchanged by `curves/2` into `spirals.tex`, becoming thereby L^AT_EX comment lines. `curves/2` uses `gen_command/6` to generate the commands specifying the curves, here the four spirals.

Define the predicate `curves/2`!

Hint. Use `apply/2` to call a predicate whose name is known at runtime only. For example, in the query below, after defining the predicate `pol/5` the variable `Pred` is unified with `pol(4, 3, 2, 10, Y)` and then the goal `pol(4, 3, 2, 10, Y)` is satisfied via the call `apply(Pred, [])`.

```

?- consult(user).
|: pol(A, B, C, X, Y) :- Y is A + B * X + C * X * X.
|: Ctrl+D
% user://1 compiled 0.01 sec, 392 bytes
Yes
?- Pred = pol(4, 3, 2, 10, Y), apply(Pred, []).
Pred = pol(4, 3, 2, 10, 234)
Y = 234
Yes

```

Exercise 4.7. Embed the predicate `curves/2` from Exercise 4.6 into a LINUX shell script called ‘`curves`’ for creating a L^AT_EX file for defining parametric curves. The shell script will use two arguments corresponding to those of `curves/2`. (This solution will have the benefit of the underlying Prolog application remaining hidden

from the user.)

Illustrative Example.

Running the script `curves` as shown in Fig. 4.24 will have the same effect as applying the predicate `curves/2` in step ② of Exercise 4.6. The file `spirals.tex` created thereby was copied by means of the last line of Fig. 4.24

```
csenki@linux:~/scripts> ./curves spirals spirals.tex
% /home/csenki/scripts/draw.pl compiled 0.00 sec, 11,800 bytes
Input file : 'spirals'
Output file: 'spirals.tex'
LaTeX source 'spirals.tex' created
csenki@linux:~/scripts> cp spirals.tex ~/texmatter/ventus
```

Figure 4.24: Running the Shell Script `curves`

into a directory where all $\text{\LaTeX}$ source for the present document is kept. (This copy was made subsequently part of the $\text{\LaTeX}$ source by writing `\include{spirals}` in the source's top level file.) ■

Appendix A

Solutions of Selected Exercises

A.1 Chapter 1 Exercises

All Prolog source code for Chap. 1 is available in the file `enigma.pl`.

Exercise 1.1. We first disassemble the list and then assemble the reduced list by leaving out one element:

```
remove_one(List,E,Reduced) :- append(Front,[E|Back],List),
                               append(Front,Back,Reduced).
```

Exercise 1.2. Define

```
var_matrix(Size,M) :- repeat(Size,Size,RowLengths),
                      maplist(var_list,RowLengths,M).
```

with the predicate `repeat/3`,

```
repeat(X,1,[X]) :- !.
repeat(X,N,[X|R]) :- NewN is N - 1,
                     repeat(X,NewN,R).
```

for producing lists with the same entry repeated a specified number of times.

Exercise 1.3. We show three approaches. The first is, as originally suggested, by recursion.

```
list_permute([],_,[]).
list_permute([P1|Rest],L,[H|T]) :- nth1(P1,L,H),
                                   list_permute(Rest,L,T).
```

An alternative definition uses `bagof/3`.

```
?- Perm = [3,1,2], L = [_R1,_R2,_R3], bagof(_E,_I^(member(_I,Perm), nth1(_I,L,_E)),P).
Perm = [3, 1, 2]
L = [_G642, _G645, _G648]
P = [_G648, _G642, _G645]
```

Finally, we may use `maplist/3` as indicated by the query below.

```
?- dynamic(nth1_new/3), retractall(nth1_new(_,_,_)), assert(nth1_new(_L,_I,_E) :- nth1(_I,_L,_E)),
   Perm = [3,1,2], L = [_R1,_R2,_R3], maplist(nth1_new(L),Perm,P).
Perm = [3, 1, 2]
L = [_G1122, _G1125, _G1128]
P = [_G1128, _G1122, _G1125]
```

Exercise 1.4. The predicate *col/3*, defined by

```
col(Matrix,N,Column) :- maplist(nth1(N),Matrix,Column).
```

returns a specified column of a matrix as a list. We now assemble the transposed matrix *T* as the list of the columns of the original matrix *M*.

```
transpose(M,T) :- [H|_] = M,           % get H to measure NCols
                  length(H,NCols),
                  bagof(N,between(1,NCols,N),L),
                  maplist(col(M),L,T).
```

Exercise 1.5. The predicate *notin/2*, defined by

```
notin(_,[ ]).
notin(E,[H/T]) :- E \== H, notin(E,T).
```

succeeds if the first argument is not equivalent to any of the list entries. *distinct/1* is defined by recursion using *notin/2*.

```
distinct([_]).
distinct([H|T]) :- notin(H,T), distinct(T).
```

Exercise 1.6. We first define *retain_var(+Var,+VarList,-List)* by

```
retain_var(_,[],[]).
retain_var(V,[H|T],[H|L]) :- H == V, retain_var(V,T,L).
retain_var(V,[H|T],L) :- H \== V, retain_var(V,T,L).
```

It will be used as an auxiliary predicate where *List* will contain as many copies of *Var* as there are in *VarList*. For example,

```
?- retain_var(_B,[_A,_B,_A,_C,_B,_A],L).
L = [_G357, _G357]
```

Now, count the number of entries in *List*.

```
count_var(VarList,Var,Num) :- retain_var(Var,VarList,List),
                               length(List,Num).
```

An alternative, more concise (one clause) solution is suggested by the query

```
?- bagof(_E,(member(_E,[_A,_B,_A,_C,_B,_A]), _E == _A),_L),
   length(_L,N).
N = 3
```

Exercise 1.7. We define *zip/3* by recursion.

```
zip([],_,[]) :- !.
zip(_,[],[]) :- !.
zip([H1|T1],[H2|T2],[(H1,H2)|T]) :- zip(T1,T2,T).
```

The input lists need not be of the same length in which case the excess tail section of the longer one will be ignored.

Exercise 1.8. Define *total/2* by

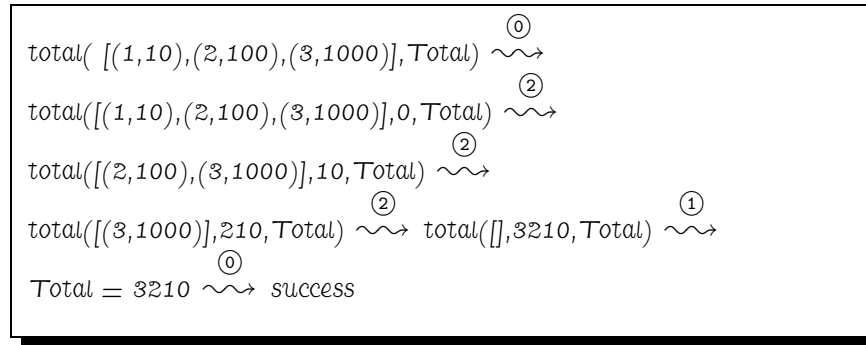
```
total(IntPairs>Total) :- total(IntPairs,0>Total). % clause 0

total([],S,S). % clause 1
total([(X,Y)|T],Acc,S) :- NewAcc is Acc + X * Y, % clause 2
                           total(T>NewAcc,S).
```

The corresponding annotated hand computations are shown in Fig. A.1.

Exercise 1.9. We first define *write_ilst(+Width,+List)* by

```
write_ilst(Width, List) :- length(List,Length),
                           int_to_atom(Width,WidthA),
                           concat_atom(['%',WidthA,'r'],Atom),
                           repeat(Atom,Length,Format1),
                           append(Format1,[' '],Format2),
                           concat_atom([' '|Format2],Format),
                           writef(Format,List).
```

Figure A.1: Hand Computations for *total/2*

for displaying an integer *list* in the right justified fashion. *Width* takes the number of digits reserved for the display of each entry. For example,

```
?- write_elist(8, [12, 345, 6789]).
[      12      345      6789]
```

(*repeat/2* has been taken from the solution of Exercise 1.2, p. 161.)

The matrix is finally displayed row-wise by

```
write_matrix(Matrix) :- largest(Matrix,Max),
                        ndigits(Max,ND),
                        Width is ND + 2,
                        write_matrix(Width,M).
```

using the predicates

- *largest(+Matrix,-Max)* for calculating the largest entry of *Matrix* (definition not shown here),
- *ndigits/2* for calculating the number of digits of a number is defined in terms of *digits/2* by

```
ndigits(N,ND) :- digits(N,D), length(D,ND).
```

(*digits/2* was defined in Exercise 4.8 of [9, p. 136] to return the list of digits of an integer; see also [9, pp. 173–174].)

- *write_matrix/2* with

```
write_matrix(_, []).
write_matrix(Width, [H|T]) :- write_elist(Width, H), nl,
                              write_matrix(Width, T).
```

Exercise 1.10. The completed Table 1.3 is shown as Table A.1. As the full definition of *next_partition/2* is available in *enigma.pl*, we want to elaborate on one particular case only, typified by the fifth column in Table A.1. The Ferrers diagrams of the ‘current’ and ‘next’ partition are shown in Fig. A.2, part (a) and (b), respectively. We proceed as follows.

<i>Current Partition</i>	$[2^3 4^1 6^2]$	$[4^1 6^3]$	$[4^3 5^2]$	$[1^3 2^4 3^1 4^2]$
<i>Next Partition</i>	$[1^2 2^2 4^1 6^2]$	$[1^1 3^1 6^3]$	$[1^1 3^1 4^2 5^2]$	$[1^5 2^3 3^1 4^2]$
<i>Step Used</i>	(i)	(i)	(i)	(ii)

<i>Current Partition</i>	$[1^5 5^1 6^2]$	$[1^3 5^1 7^2]$	$[1^5 4^3 5^1]$
<i>Next Partition</i>	$[2^1 4^2 6^2]$	$[4^2 7^2]$	$[3^3 4^2 5^1]$
<i>Step Used</i>	(ii)	(ii)	(ii)

Table A.1: Partitions

- We unify the current partition's list representation with $[(1, A), (K, 1) / T]$. (The group of sixes will, since they remain unchanged, be subsumed in the list's tail.)
- The total number of marked tokens is $A + L$. They are to form as many groups of size $L - 1$ as possible. The number of them will be computed by integer division ($//$). The leftovers form the bottom row of the

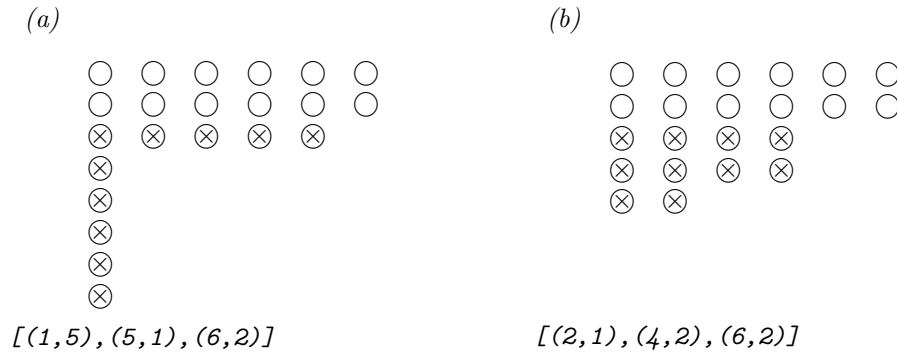


Figure A.2: Ferrers Diagrams and their Prolog Representations

new Ferrers diagram. The number of them is the division's remainder (Prolog's *mod*).

- These ideas give rise to the following clause.

```
next_partition([(1,A),(L,1)|T],[(Rest,1),(NewL,Rat)|T]) :- L > 2,
    NewL is L - 1,
    Rest is (A + L) mod NewL,
    Rest > 0,
    Rat is (A + L) // NewL.
```

Exercise 1.11. Define *next_int/3* by

```
next_int(High,I,NextI) :- succ(I,NextI), NextI =< High.
```

and use it as

```
?- generator(next_int(9),3,I).
I = 3 ;
I = 4 ;
...
I = 9 ;
No
```

(This is in effect a new implementation of the built-in predicate *between/3* [9, p. 41].)

Exercise 1.12. The horizontal and vertical transitions in Fig. 1.6 are encoded by

```
next_pair((0,0),(0,1)) :- !.
next_pair((0,N),(0,NextN)) :- even(N), succ(N,NextN), !.
next_pair((M,0),(NextM,0)) :- odd(M), succ(M,NextM), !.
```

where *even/1* and *odd/1* are respectively defined by

```
even(N) :- 0 is N mod 2. odd(N) :- 1 is N mod 2.
```

The built-in conditional *->/2* [9, p. 91] may be used to implement the diagonal transitions in Fig. 1.6.

```

?- current_predicate(Pred,_), atom_prefix(Pred,'temp').
No
?- tmp_predname(_Temp), _Term =.. [_Temp,(_I,_I)], assert(_Term).
Yes
?- current_predicate(Pred,_), atom_prefix(Pred,'temp').
Pred = temp_0 ;
No
?- tmp_predname(_Temp), _Term =.. [_Temp,(_I,_I)], assert(_Term).
Yes
?- current_predicate(Pred,_), atom_prefix(Pred,'temp').
Pred = temp_1 ;
Pred = temp_0 ;
No

```

Figure A.3: Creating Distinct Temporary Predicate Names

```

next_pair((M,N),(NextM,NextN)) :- Sum is M + N,
                                (odd(Sum) -> succ(M,NextM), succ(NextN,N);
                                succ(NextM,M), succ(N,NextN)), !.

```

Pairs starting with $(1,1)$, say, are generated by

```

?- generator(next_pair,(1,1),P).
P = 1, 1 ;
P = 0, 2 ;
P = 0, 3 ;
P = 1, 2 ;
...

```

Exercise 1.13. *tmp_predname/1* returns, each time it is invoked, an atom for naming a temporary predicate.

```

tmp_predname(Temp) :- int(0,N),
                      int_to_atom(N,Tag),
                      concat_atom(['temp_',Tag],Temp),
                      not(current_predicate(Temp,_)), !.

```

The interactive session in Fig. A.3 illustrates how *tmp_predname/1* may be used to produce predicate names hitherto not present in the database. (See also inset.) In the definition of the new version of *generator/3*, its structure is retained except that now the goals (terms) referring to the temporary predicate are constructed using the built-in predicate *univ* ($=..$) [9, p. 43].

Built-in Predicate: *atom_prefix(+Atom,+Prefix)*

Succeeds if the second argument is a *Prefix* to the *Atom* in the first argument.
Example:

?- atom_prefix(software,soft).

Yes

?- atom_prefix(software,war).

No

```

generator2(Pred,From,Elem) :- tmp_predname(TempName),
                             Term1 =.. [TempName,First,First],
                             Term2 =.. [TempName,Last,E],
                             Term3 =.. [TempName,New,E],
                             Term4 =.. [TempName,From,Elem],
                             assert(Term1),
                             assert(Term2 :- (call(Pred,Last,New), Term3)),
                             write('Defined '),
                             write(TempName),
                             write('/2 in the database.\n'),
                             Term4.

```

(Lines reporting new predicates' names have been included.) We now use the new version of *generator/3* to define a new version of *pairs/1* by

```

pairs2((I,J)) :- generator2(succ,0,Sum),
                 generator2(next_int(Sum),0,I),
                 J is Sum - I.

```

It will behave on backtracking as intended:

```

?- pairs2(P).
Defined temp_0/2 in the database.
Defined temp_1/2 in the database.
P = 0, 0 ;
Defined temp_2/2 in the database.
P = 0, 1 ;
P = 1, 0 ;
Defined temp_3/2 in the database.
P = 0, 2 ;
P = 1, 1 ;
...

```

We may wish to remove all unwanted temporary predicates from the database. This is accomplished by the following failure driven loop.

```

?- current_predicate(Pred,_), atom_prefix(Pred,'temp_'), Term =.. [Pred,'_', '_'], retractall(Term), fail.
No

```

The query below finally confirms that no predicate of arity 2 whose name starts with 'temp_' is left in the database.

```

?- current_predicate(Pred,_), atom_prefix(Pred,'temp_'), atom_concat(Pred,'/2',P)1, listing(P), fail.
ERROR: No predicates for 'temp_1/2'
ERROR: No predicates for 'temp_0/2'
ERROR: No predicates for 'temp_3/2'
ERROR: No predicates for 'temp_2/2'
No

```

Exercise 1.14. Based on the annotated hand computations in Fig. A.4, p. 170, the predicate *split/4* is defined in (P-A.1).

¹We have met *atom_concat/3* in [9, p. 138].

$\text{split}([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16], [(2,1),(3,3),(5,1)], [], S) \rightsquigarrow^{\textcircled{3}}$
 $\text{split}([3,4,5,6,7,8,9,10,11,12,13,14,15,16], [(2,0),(3,3),(5,1)], [[1,2]], S) \rightsquigarrow^{\textcircled{2}}$
 $\text{split}([3,4,5,6,7,8,9,10,11,12,13,14,15,16], [(3,3),(5,1)], [[1,2]], S) \rightsquigarrow^{\textcircled{3}}$
 $\text{split}([6,7,8,9,10,11,12,13,14,15,16], [(3,2),(5,1)], [[3,4,5], [1,2]], S) \rightsquigarrow^{\textcircled{3}}$
 $\text{split}([9,10,11,12,13,14,15,16], [(3,1),(5,1)], [[6,7,8], [3,4,5], [1,2]], S) \rightsquigarrow^{\textcircled{3}}$
 $\text{split}([12,13,14,15,16], [(3,0),(5,1)], [[9,10,11], [6,7,8], [3,4,5], [1,2]], S) \rightsquigarrow^{\textcircled{2}}$
 $\text{split}([12,13,14,15,16], [(5,1)], [[9,10,11], [6,7,8], [3,4,5], [1,2]], S) \rightsquigarrow^{\textcircled{3}}$
 $\text{split}([], [(5,0)], [[12,13,14,15,16], [9,10,11], [6,7,8], [3,4,5], [1,2]], S) \rightsquigarrow^{\textcircled{1}}$
 $\text{reverse}([[12,13,14,15,16], [9,10,11], [6,7,8], [3,4,5], [1,2]], S) \rightsquigarrow$
 $S = [[1,2], [3,4,5], [6,7,8], [9,10,11], [12,13,14,15,16]] \rightsquigarrow \text{success}$

Figure A.4: Annotated Hand Computations for *split/4*

Prolog Code P-A.1: Definition of `split/4`

```

1 split([],[(_,0)],Acc,S)           :- reverse(Acc,S), !. % clause 1
2 split(L,[(_,0)|T],Acc,S)        :- split(L,T,Acc,S). % clause 2
3 split(L,[(K,AlphaK)|T],Acc,S) :- % clause 3
4     AlphaK > 0, %
5     append(L1,L2,L), %
6     length(L1,K), %
7     NewAlphaK is AlphaK - 1, %
8     split(L2,[(K,NewAlphaK)|T], [L1|Acc],S). %

```

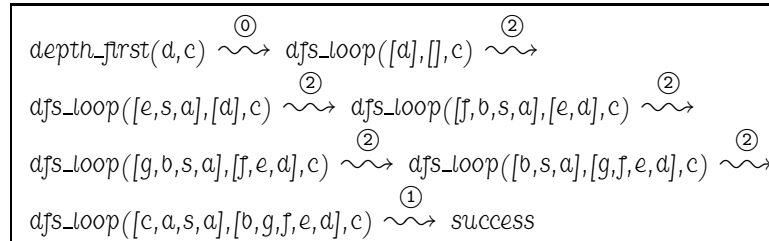
(Notice the concise way *L1* is declared to be the front part of *L* with a specific length.)

A.2 Chapter 2 Exercises

All Prolog source files for Chap. 2 are available in the directory `plsearch`.

Exercise 2.2, part (a). Add to the database in Fig. 2.2 the facts

```
connect(u,v). connect(u,w). connect(v,w).
```

Figure A.5: Hand Computations for the Query `?- depth_first(d,c)`.

Part (b). The successor nodes used in the hand computations for the query `?- depth_first(d,c)`. (Fig. A.5) may be gleaned from Fig. 2.4, p. 50. The interactive session in Fig. A.6, p. 173, confirms the hand computations. The hand computations for the query `?- depth_first(u,c)` are shown in Fig. A.7, p. 173. (The tree in Fig. A.8, p. 173, drawn by inspecting the database, may be used to work out successor nodes.) They are confirmed by the query in Fig. A.9, p. 174. The query in Fig. A.9 illustrates a perhaps unexpected feature of our implementation: it is possible for a node to be open and closed at the same time. (Algorithm 2.3.2 does not check for this condition.)

Exercise 2.3. We consider two possibilities. The first definition in (P-A.2) uses `maplist/3`.

Prolog Code P-A.2: First definition of `extend_path/3`

```

1 extend_path(Nodes,Path,ExtendedPath) :-
2     maplist(glue(Path),Nodes,ExtendedPath).
3 glue(T,H,[H|T]).
```

The auxiliary predicate `glue/3` in (P-A.2) is for ‘glueing’ head and tail together. (The order of arguments of `glue/3` is chosen so as to facilitate *partial application* of `glue/3` in (P-A.2) by fixing its first argument.) In (P-A.3) another definition of `extend_path/3` is shown. It uses recursion.

Prolog Code P-A.3: Second definition of `extend_path/3`

```

1 extend_path([],_,[]). % clause 1
2 extend_path([Node|Nodes],Path,[[Node|Path]|Extended]) :- % clause 2
3     extend_path(Nodes,Path,Extended). %
```

We shall be working with (P-A.3) in the main body of the text.

Exercise 2.4. For the new connectivity, add the clause

`connect(b,s).`

to the file `links.pl`.

The new version of `is_path/1` (in the file `searchinfo.pl`) will be formulated as a *negation*, i.e.

```

?- consult(df2).
% links compiled into edges 0.00 sec, 1,900 bytes
% df2 compiled 0.05 sec, 3,892 bytes
Yes
?- depth_first(d,c).
Open: [d], Closed: []
Node d is being expanded. Successors: [e, s, a]
Open: [e, s, a], Closed: [d]
Node e is being expanded. Successors: [f, b, d]
Open: [f, b, s, a], Closed: [e, d]
Node f is being expanded. Successors: [g, e]
Open: [g, b, s, a], Closed: [f, e, d]
Node g is being expanded. Successors: [h]
Open: [h, b, s, a], Closed: [g, f, e, d]
Node h is being expanded. Successors: [c, e, a]
Open: [c, a, s, a], Closed: [h, g, f, e, d]
Goal found: c
Yes

```

Figure A.6: Interactive Session for the Query `?- depth_first(d,c).`

```

depth_first(u,c)  $\rightsquigarrow$  dfs_loop([u],[],c)  $\rightsquigarrow$ 
dfs_loop([v,w],[u],c)  $\rightsquigarrow$  dfs_loop([w,w],[v,u],c)  $\rightsquigarrow$ 
dfs_loop([w],[w,v,u],c)  $\rightsquigarrow$  dfs_loop([], [w,w,v,u],c)  $\rightsquigarrow$  failure

```

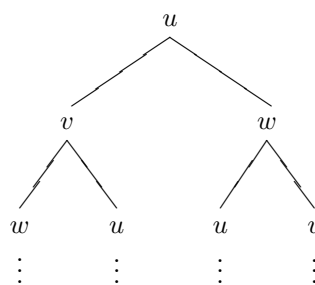
Figure A.7: Hand Computations for the Query `?- depth_first(u,c).`

Figure A.8: Tree for Finding Successor Nodes in the New Component

```
?- depth_first(u,c).  
Open: [u], Closed: []  
Node u is being expanded. Successors: [v, w]  
Open: [v, w], Closed: [u]  
Node v is being expanded. Successors: [w, u]  
Open: [w, w], Closed: [v, u]  
Node w is being expanded. Successors: [u, v]  
Open: [w], Closed: [w, v, u]  
Node w is being expanded. Successors: [u, v]  
Open: [], Closed: [w, w, v, u]  
No
```

Figure A.9: Interactive Session for the Query `?- depth_first(u,c).`

```
is_path(L) :- not(prohibit(L)).
```

with *prohibit/1* specifying the conditions which a path *must not* have.

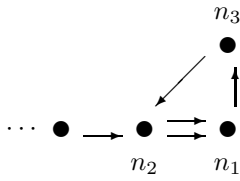
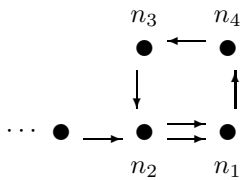
Example Path	Prolog Clause
	$\text{same}([N1, N2, N3, N1, N2 _]).$
	$\text{same}([N1, N2, N3, N4, N1, N2 _]).$

Table A.2: Example Paths and Prolog Implementations – Case One

- Not allowed is a path whose *leading* edge is the *same* as some other edge in its tail (see Table A.2). This condition is implemented by

```
same([N1, N2, _, N1, N2 | _]).
same([N1, N2, _ | T]) :- same([N1, N2 | T]).
```

- Not allowed is a path whose *leading* edge is *opposite* to some other edge in its tail (see Table A.3). This condition is implemented by

```
opposite([N1, _, N1 | _]).
opposite([N1, N2, _, _, N2, N1 | _]).
opposite([N1, N2, N3, N4, _ | T]) :- opposite([N1, N2, N3, N4 | T]).
```

It is seen by an inductive argument that if the above two conditions are observed, no path with repeated edges will ever be constructed by the search algorithm. Concentrating on the leading edge therefore does not pose a restriction but simplifies the implementation. Define now *prohibit/1* in *searchinfo.pl* by

```
prohibit(L) :- same(L).
prohibit(L) :- opposite(L).
```

The new version of *depth_first/4* will behave as illustrated in Fig. A.10, p. 176.

Exercise 2.5. The new version will be placed in the same file as the old one (viz *df.pl*). We start by defining a new version of *extend_path/3*, called *extend_path_dl/3*, as shown in Fig. A.11, p. 177.

This is a straightforward ‘translation’ of *extend_path/3* and it behaves as follows,

```
?- extend_path_dl([f, d], [e, b, a, s], L3-L1).
L3 = [[f, e, b, a, s], [d, e, b, a, s] | _G361]
L1 = _G361 ;
No
```

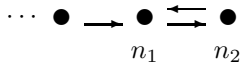
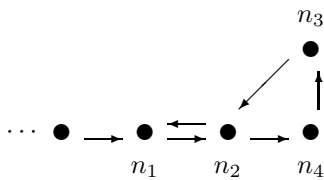
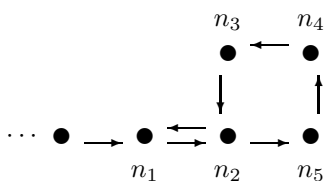
Example Path	Prolog Clause
	<code>opposite([N1,N2,N1 _]).</code>
	<code>opposite([N1,N2,N3,N4,N2,N1 _]).</code>
	<code>opposite([N1,N2,N3,N4,N5,N2,N1 _]).</code>

Table A.3: Example Paths and Prolog Implementations – Case Two

```

?- consult(df4).
% links compiled into edges 0.00 sec, 1,964 bytes
% searchinfo compiled into info 0.00 sec, 2,120 bytes
% df4 compiled 0.05 sec, 6,272 bytes
Yes
?- depth_first(s,goal_path,link,Path).
Path = [s, a, b, e, f, g] ;
Path = [s, a, b, s, d, e, f, g] ;
Path = [s, a, d, e, f, g] ;
Path = [s, a, d, s, b, e, f, g] ;
Path = [s, d, e, f, g] ;
Path = [s, d, a, b, e, f, g] ;
Path = [s, d, a, s, b, e, f, g] ;
Path = [s, b, e, f, g] ;
Path = [s, b, a, d, e, f, g] ;
Path = [s, b, a, s, d, e, f, g] ;
No

```

Figure A.10: Sample Session for *depth_first/4*

```

extend_path_dl([],_,E-E).
extend_path_dl([N|Ns],Path,[N|Path]|E1)-E2) :-
    extend_path_dl(Ns,Path,E1-E2).

```

Figure A.11: Definition of *extend_path_dl/3*

In the same fashion, direct translation of the two clauses of *dfs_loop/4* from Fig. 2.15, p. 65, gives the clauses shown in Fig. A.12, p. 178. (Notice that, as intended, the *append* goal has been dispensed with. Also notice that the new clauses won't interfere with the old ones and we may place them in the same file.) Fig. A.13, p. 178, illustrates the updating of the agenda by this new version of *dfs_loop/4*.

The new version of *depth_first/4* is shown in (P-A.4).

Prolog Code P-A.4: Definition of *depth_first_dl/4*

```

1 depth_first_dl(Start,G_Pred,C_Pred,PathFound) :-
2     dfs_loop([[Start]|L]-L,G_Pred,C_Pred,PathFoundRev),
3     reverse(PathFoundRev,PathFound).

```

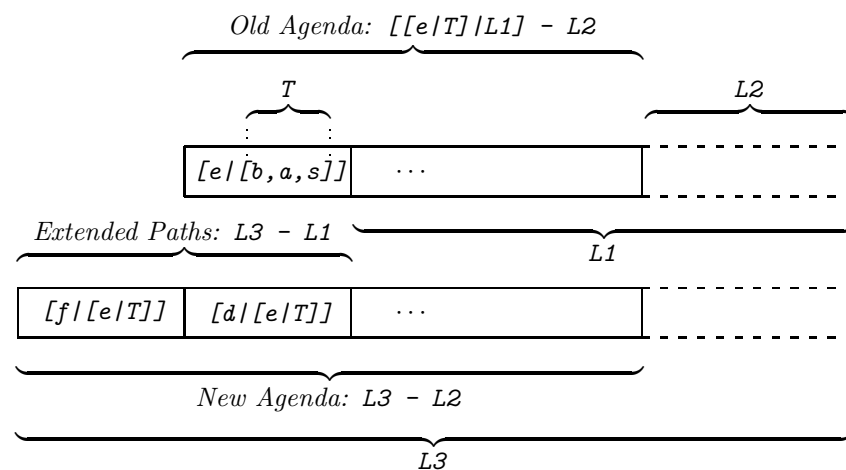
```

dfs_loop([Path|_]--,G_Pred,_,Path) :- call(G_Pred,Path).

dfs_loop([[CurrNode|T] | L1]-L2,G_Pred,C_Pred,PathFound) :-
    successors(C_Pred,CurrNode,SuccNodes),
    findall(Node,(member(Node,SuccNodes),
                  is_path([Node,CurrNode|T])),Nodes),
    extend_path_d1(Nodes,[CurrNode|T],L3-L1),
    dfs_loop(L3-L2,G_Pred,C_Pred,PathFound).

```

LIFO updating
of the agenda

Figure A.12: New Clauses for *dfs_loop/4*Figure A.13: Updating of the Agenda by *dfs_loop/4*

It is seen that on backtracking *depth_first/4* does not quite behave as expected:

```
?- depth_first_dl(s,goal_path,link,Path).
Path = [s, a, b, e, f, g] ;
Path = [s, a, d, e, f, g] ;
Path = [s, d, e, f, g] ;
Path = [s, d, a, b, e, f, g] ;
Path = [g] ;
Path = [_G2571, g] ;
...
```

What is the explanation for the spurious solutions and non-termination, and, what is the remedy? The search should finish once the agenda is empty. In the old version based on ordinary lists, *dfs_loop/4* terminates by failure if its first argument is unified with the empty list:

```
?- dfs_loop([],goal_path,link,Path).
No
```

As *L-L* stands for the empty list, the corresponding query would be

```
?- dfs_loop(L-L,goal_path,link,Path).
L = [[g|_G415]|_G412]
Path = [g|_G415] ;
...
```

It succeeds, however. To prevent this from happening, we add in front of all other clauses of *dfs_loop/4* to the database the clause

```
dfs_loop(L-L,_,_,_) :- !, fail.
```

upon which, as expected, the above query will fail:

```
?- dfs_loop(L-L,goal_path,link,Path).
No
```

Unfortunately, though, *depth_first_dl/4* now *always* fails:

```
?- depth_first_dl(s,goal_path,link,Path).
No
```

To see why, we first rewrite the new clause in the form

```
dfs_loop(A-A, B, C, D) :- !, fail.
```

The last query tries first to satisfy the subgoal

```
dfs_loop([[Start]/L]-L,G-Pred,C-Pred,PathFoundRev)
```

with *Start* = *s*, *G-Pred* = *goal_path*, *C-Pred* = *link* and *PathFoundRev* = *Path*. The added new clause will now be tried first. In particular, it will be attempted to unify its first argument with *[[s]/L]-L*. Unification should *not* succeed simply because *[[s]/L]-L* does not stand for the empty list. Let's explore interactively what really happens:

```
?- A-A = [[s]/L]-L.
A = [[s], [s], [s], [s], [s], [s], [s], [s], [...]|...]
L = [[s], [s], [s], [s], [s], [s], [s], [s], [...]|...]
Yes
```

It is seen that matching succeeds because Prolog does not check whether unification will give rise to an infinite term (due to the same variable occurring in both terms to be unified).² Unification of these terms will fail, however, if we use *unify_with_occurs_check/2*, an SWI-Prolog implementation of full unification:

```
?- unify_with_occurs_check(A-A, [[s]/L]-L).
No
```

²In the above query, essentially, unification of *[[s]/L]* and *L* is attempted. This *should* fail. However, without an *occurs check* Prolog reports success:

```
?- [[s]/L] = L.
L = [[s], [s], [s], [s], [s], [s], [s], [s], [...]|...]
Yes
```

Built-in Predicate: `unify_with_occurs_check(?Term1, ?Term2)`

Unifies the two terms *Term1* and *Term2* just as `=/2` would do. If, however, using `=/2` would give rise to an infinite term, `unify_with_occurs_check/2` will fail. Example:

```
?- unify_with_occurs_check(f(X,a),f(a,X)).
X = a
Yes
?- X = f(X).
X = f(f(f(f(f(f(f(f(...))))))))))
Yes
?- unify_with_occurs_check(X,f(X)).
No
```

In the added clause (P-A.5), this implementation of unification is therefore used.

Prolog Code P-A.5: Additional clause of `dfs_loop/4`

```
1 dfs_loop(L1-L2,_,_,_) :- unify_with_occurs_check(L1,L2), !, fail.
```

Prolog now responds as expected:

```
?- consult(df).
% links compiled into edges 0.00 sec, 1,900 bytes
% searchinfo compiled into info 0.00 sec, 1,016 bytes
Warning: (c:/prolog/plsearch/df.pl:34):
  Clauses of dfs_loop/4 are not together in the source-file3
% df compiled 0.00 sec, 6,272 bytes
Yes
?- depth_first_dl(s,goal_path,link,Path).
Path = [s, a, b, e, f, g] ;
Path = [s, a, d, e, f, g] ;
Path = [s, d, e, f, g] ;
Path = [s, d, a, b, e, f, g] ;
No
```

The only drawback of `unify_with_occurs_check/2` is that it is computationally more expensive than the predicate `=/2`.

The computational advantage of the difference list based version is confirmed by

```
?- time(findall(_P,depth_first_dl(s,goal_path,link,_P),_Ps)).
% 1,293 inferences in 0.00 seconds (Infinite Lips)
Yes
?- time(findall(_P,depth_first(s,goal_path,link,_P),_Ps)).
% 1,414 inferences in 0.06 seconds (23567 Lips)
Yes
```

³To suppress this warning message, place the directive

```
:- discontinuous dfs_loop/4.
just after the use_module directives in df.pl.
```

```

:- disjoint dfs_loop/4.
...
breadth_first_dl(Start,G_Pred,C_Pred,PathFound) :-
    bfs_loop([[Start]|L]-L,G_Pred,C_Pred,PathFoundRev),
    reverse(PathFoundRev,PathFound).

bfs_loop(L1-L2,_,_,_) :- unify_with_occurs_check(L1,L2), !, fail.
bfs_loop([Path|_]-_,G_Pred,_,Path) :- call(G_Pred,Path).
bfs_loop([CurrNode|T]|L1-L2,G_Pred,C_Pred,PathFound) :-
    successors(C_Pred,CurrNode,SuccNodes),
    findall(Node,(member(Node,SuccNodes),
        is_path([Node,CurrNode|T])),Nodes),
    extend_path_dl(Nodes,[CurrNode|T],L2-L3),
    bfs_loop(L1-L3,G_Pred,C_Pred,PathFound).

% auxiliary predicates ...
...
extend_path_dl([],_,E-E).
extend_path_dl([N|Ns],Path,[N|Path]|E1-E2) :-
    extend_path_dl(Ns,Path,E1-E2).

```

*Copied from the augmented version of df.pl
(Exercise 2.5, Fig. A.11, p. 177)*

FIFO updating of the agenda

Figure A.14: Clauses Added to bf.pl

Exercise 2.6. The clauses added to bf.pl are shown in Fig. A.14. The new version responds as intended:

```

?- breadth_first_dl(s,goal_path,link,Path).
Path = [s, d, e, f, g] ;
Path = [s, a, b, e, f, g] ;
Path = [s, a, d, e, f, g] ;
Path = [s, d, a, b, e, f, g] ;
No

```

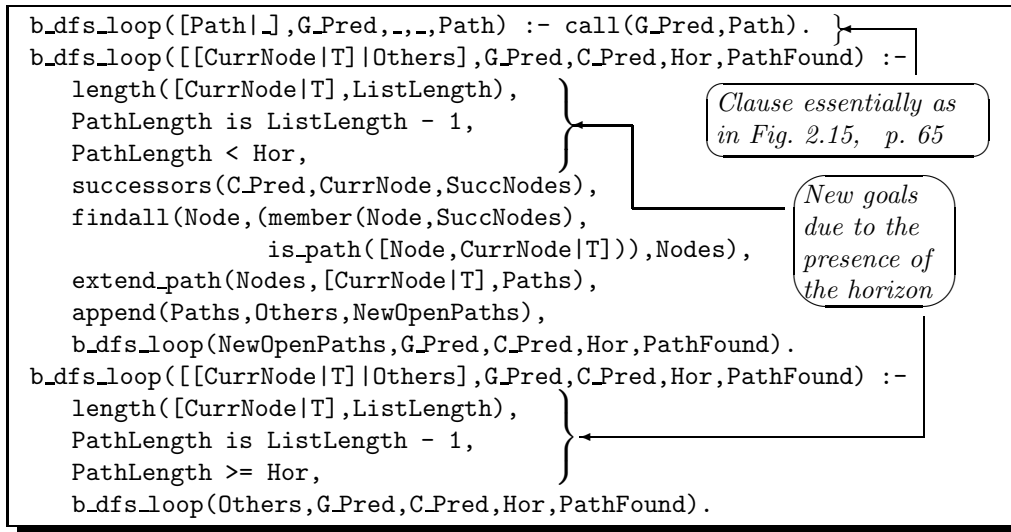
And, it performs better than the old one:

```

?- time(findall(_P,breadth_first_dl(s,goal_path,link,_P),_Ps)).
% 1,293 inferences in 0.00 seconds (Infinite Lips)
Yes
?- time(findall(_P,breadth_first(s,goal_path,link,_P),_Ps)).
% 1,378 inferences in 0.00 seconds (Infinite Lips)
Yes

```

Exercise 2.7. See Fig. A.15.

Figure A.15: Definition of *b_dfs_loop/5* (Exercise 2.7)

```

:- use_module(bdf).
:- dynamic(lastdepth/1).

iterative_deepening(Start,G_Pred,C_Pred,PathFound) :-
    retractall(lastdepth(_)),
    assert(lastdepth(0)),
    iterative_deepening_aux(1,Start,G_Pred,C_Pred,PathFound).

iterative_deepening_aux(Depth,Start,G_Pred,C_Pred,PathFound) :-
    bounded_df(Start,G_Pred,C_Pred,Depth,PathFound).
iterative_deepening_aux(Depth,Start,G_Pred,C_Pred,PathFound) :-
    retractall(lastdepth(_)),
    assert(lastdepth(Depth)),
    NewDepth is Depth + 1,
    iterative_deepening_aux(NewDepth,Start,G_Pred,C_Pred,PathFound).

```

← Declare *lastdepth/1* to be dynamic
 ← Initialize saved value of horizon to zero
 ← Saving old value of horizon

Figure A.16: Modified Version of `iterd.pl` (Exercise 2.8)

Exercise 2.8. We add four new goals to the first clause of `b_dfs_loop/5`; this is shown in (P-A.6).

Prolog Code P-A.6: Modified first clause of `b_dfs_loop/5`

```

1 b_dfs_loop([Path|_],G_Pred,_,_,Path) :- call(G_Pred,Path),
2                                         lastdepth(LastDepth),
3                                         length(Path,ListLength),
4                                         PathLength is ListLength - 1,
5                                         PathLength > LastDepth.

```

Furthermore, we need to modify `iterd.pl` which is shown in Fig. A.16.

Exercise 2.9. To have a unique solution, add the cut (!) in the definition of `iterative_deepening/4` as follows.

```

iterative_deepening(Start,G_Pred,C_Pred,PathFound) :-
    iterative_deepening_aux(1,Start,G_Pred,C_Pred,PathFound), !.

```

Exercise 2.14. Let us assume that we have consulted `loop_puzzle1a.pl`; then, `automated.pl` will also be loaded. The predicate `segment/1` may be defined interactively by

```

?- consult(user).
|: segment(S) :- (circle(P); sharp(P)), link([P],S).
|: Ctrl D
% user compiled 61.14 sec, 332 bytes
Yes

```

It will generate all segments for the particular problem:

```

?- segment(S).
S = [pos(2,1), pos(1,1), pos(1,2), pos(1,3)] ;

```

```
S = [pos(2,2), pos(1,2), pos(1,3)] ;
...
```

All pairs of linked segments may be generated thus

```
?- segment(S1), link(S1,S2).
S1 = [pos(2,1), pos(1,1), pos(1,2), pos(1,3)] S2 = [pos(2,2)] ;
...
```

This generator may be used to define a new version of *link/2* by *facts*. (We can do this because the network and therefore the number of facts is finite.) We do this by a failure driven loop:

```
?- segment(S1), link(S1,S2), assert(newlink(S1,S2)), fail.
No
?- listing(newlink).
newlink([pos(2,1), pos(1,1), pos(1,2), pos(1,3)], [pos(2,2)]).
...
```

Use now *newlink/2* as you would use *link/2*.

The number of nodes and number of directed edges are respectively found by

```
?- setof(_S,segment(_S,_Ss), length(_Ss,L).
L = 37
?- setof((_S1,_S2),newlink(_S1,_S2),_Ps), length(_Ps,L).
L = 99
```

To find out the corresponding quantities for the ‘hand-knit’ solution, we first consult the file *hand_knit.pl*. Then, we enter the marks’ positions in the database, followed by a definition of *segment/1* as before:

```
?- consult(user).
|: circle(pos(1,4)). circle(pos(3,5)).
|: circle(pos(4,2)). circle(pos(6,6)).
|: sharp(pos(1,6)). sharp(pos(2,1)). sharp(pos(2,2)).
|: sharp(pos(4,1)). sharp(pos(5,5)).
|: segment(S) :- (circle(P); sharp(P)), link([P],S).
|: Ctrl+D
% user compiled 0.03 sec, 1,256 bytes
Yes
```

Whereas the number of nodes is confirmed to be 37 by exactly the same query as before, the number of edges is now found by

```
?- setof((_S1,_S2),(segment(_S1),link(_S1,_S2)),_Ps),
length(_Ps,L).4
L = 166
```

Exercise 2.19. The additional constraint requires that the length of the goal path be equal to the number of positions on the board – the board *size*. Since paths are represented as lists of segments, which themselves are lists of board positions, the path length will be the length of the path’s flattened list representation. This is implemented in (P-A.7) by adding four new goals to the definition of *goal_path/1*. (The predicate *goal_path/1*

⁴Here we have explicitly to specify *_S1* to be a segment as *link/2* has been defined in *hand_knit.pl* by using the wilde card (*_*) in its first argument. Failing to do so would instantiate *_S1* to the wildcard, returning an erroneous value for the number of network connections which, incidentally, would be the number of facts defining *link/2* in *hand_knit.pl*.

is defined in `loops.pl`.)

Prolog Code P-A.7: *Augmented definition of `goal_path/1`*

```
1 goal_path([H|T]) :- number_of_marks(M),  
2                     length([H|T],M),  
3                     last(E,T),  
4                     link(H,E),  
5                     size(Row,Col),      % added goal  
6                     Size is Row * Col,  % added goal  
7                     flatten([H|T],F),   % added goal  
8                     length(F,Size).      % added goal
```

A.3 Chapter 3 Exercises

All Prolog source files for Chap. 3 are available in the directory `plsearch`.

Exercise 3.2. *Manual solution.* We get the straight line distances from any node to node *10* by Pythagoras (Table A.4). The edge lengths for Fig. 3.4, shown in Table A.5, are obtained from the node co-ordinates in Table 3.2.

Node	1	2	3	4	5	6	7	8	9
Distance to node 10	4.00	4.24	5.83	2.00	2.24	5.39	3.16	1.41	5.10

Table A.4: Values of H

—	—	—	—	—	—	4	2	6	10
—	—	—	—	5	1	—	—	9	
—	—	—	—	1	5	—	8		
—	—	—	4	—	—	7			
—	3	1	—	—	6				
—	3	5	—	5					
—	4	6	4						
6	—	3							
4	2								
1									

Table A.5: Distances between Nodes (Edge Lengths) in Fig. 3.4

The hand computations in Fig. A.18, p. 189, tell us that the shortest route is

$$1 \rightarrow 2 \rightarrow 5 \rightarrow 8 \rightarrow 10$$

and its length is 10.

Prolog implementation. We define in `graph.b.pl` the predicates `link/2` and `in/3` with obvious meanings.

```
link(1,2). link(1,3). ...
in(1,1,4). in(2,2,7). ...
```

The heuristic is the Euclidean distance, defined by `e_cost/3` in (P-A.8).

Prolog Code P-A.8: Definition of `e_cost/3`

```
1 e_cost(Node,Goal,D) :- in(Node,X1,Y1),
2                       in(Goal,X2,Y2),
3                       D is sqrt((X1 - X2)^2 + (Y1 - Y2)^2).
```

The edge costs are calculated by the city block distance, defined by `edge_cost/3` in (P-A.9).

Prolog Code P-A.9: Definition of `edge_cost/3`

```
1 edge_cost(Node1,Node2,Cost) :- link(Node1,Node2),
2                               in(Node1,X1,Y1),
3                               in(Node2,X2,Y2),
4                               Cost is abs(X1 - X2) + abs(Y1 - Y2).
```

```
?- consult(graph_b).  
% asearches compiled into a_ida_idaeps 0.00 sec, 7,736 bytes  
% graph_b compiled 0.00 sec, 14,800 bytes  
Yes  
?- path.  
Select start node 1, ..., 10: 1.  
Select goal node 1, ..., 10: 10.  
Select algorithm (a/ida/idaeps)... a.  
% 561 inferences in 0.00 seconds (Infinite Lips)  
Solution in 4 steps.  
1 -> 2 -> 5 -> 8 -> 10  
Total cost: 10  
Yes
```

Figure A.17: Automated Search

The remaining predicates are adopted from `graph_a.pl` with minor modifications. Fig. A.17 shows the automated search.

$[4.00-[1]-0] \xrightarrow{\textcircled{1}}$
 $[8.24-[2,1]-4, 11.83-[3,1]-6] \xrightarrow{\textcircled{2}}$
 $[8.24-[2,1]-4, 11.83-[3,1]-6] \xrightarrow{\textcircled{1}}$
 $[10-[4,2,1]-8, 9.24-[5,2,1]-7, 12.39-[6,2,1]-7, 11.83-[3,1]-6] \xrightarrow{\textcircled{2}}$
 $[9.24-[5,2,1]-7, 10-[4,2,1]-8, 11.83-[3,1]-6, 12.39-[6,2,1]-7] \xrightarrow{\textcircled{1}}$
 $[9.41-[8,5,2,1]-8, 17.10-[9,5,2,1]-12, 10-[4,2,1]-8, 11.83-[3,1]-6, 12.39-[6,2,1]-7] \xrightarrow{\textcircled{2}}$
 $[9.41-[8,5,2,1]-8, 10-[4,2,1]-8, 11.83-[3,1]-6, 12.39-[6,2,1]-7, 17.10-[9,5,2,1]-12] \xrightarrow{\textcircled{1}}$
 $[10.00-[10,8,5,2,1]-10, 10-[4,2,1]-8, 11.83-[3,1]-6, 12.39-[6,2,1]-7, 17.10-[9,5,2,1]-12] \xrightarrow{\textcircled{2}}$
 $[10.00-[10,8,5,2,1]-10, 10-[4,2,1]-8, 11.83-[3,1]-6, 12.39-[6,2,1]-7, 17.10-[9,5,2,1]-12] \xrightarrow{\textcircled{3}} \text{success}$

Figure A.18: Hand Computations: The Evolution of the Agenda for the *A*-Algorithm (from node 1 to node 10 in Fig 3.4)

Exercise 3.3, part (c). We search the network in Fig. 3.6 by the interactive session in Fig. A.19.⁵

```
?- consult(graph_c).
% asearches compiled into a_ida_idaeps 0.00 sec, 7,736 bytes
% graph_c compiled 0.00 sec, 31,068 bytes
Yes
?- adj(2,_A), co_ord(2,_Co), path(_A,_Co).
Select start node 1, ..., 26: 1.
Select goal node 1, ..., 26: 26.
Select algorithm (a/ida/idaeps)... a.
% 74,926 inferences in 0.02 seconds (4795264 Lips)
Solution in 11 steps.
1 -> 2 -> 5 -> 7 -> 9 -> 11 -> 15 -> 16 -> 18 -> 21 -> 24 -> 26
Total cost: 54
Yes
```

Figure A.19: Interactive Session for Searching the Network in Fig. 3.6

Exercise 3.6. Table A.6 shows that Hill Climbing and Best First, save for the simplest of cases, do not find the shortest route to the goal node. It is also seen that Best First usually finds a shorter route to the goal node but

Test Case Number			1	2	3	4	5	6	7	8	9	10
Goal Node at Depth			8	8	10	12	13	16	16	20	30	30
Number of Moves	mp	hc	8	84	954	2200	445	444	442	348	1002	730
		bestf	8	38	262	-	91	90	88	196	-	234
	mh	hc	8	8	90	112	339	338	336	406	126	528
		bestf	8	8	10	32	45	44	42	66	74	132

Table A.6: Results for the Eight Puzzle (Hill Climbing and Best First)

at a much higher computational cost than Hill Climbing. Finally, the better heuristic (MH) is seen to deliver better solutions throughout. (Cases which could not be finished due to prohibitively long CPU times are not shown here.)

Exercise 3.11. Modify the clauses of *a_loop/3* and *dfs_contour_loop/6* by replacing each occurrence of the goal

```
findall(Node, (member(Node, SuccNodes), not(member(Node, T))), Nodes)
```

by

```
findall(Node, member(Node, SuccNodes), Nodes)
```

(The modified code is in *msearches.pl*.) Thus, for example, the gain in CPU time is 17% for case 4 with Iterative Deepening *A** and the Euclidean heuristics.

⁵The present search problem happens also to be of the type considered in Sect. 3.4. The result in Fig. A.19 is confirmed by Fig. 3.10, p. 122.

A.4 Chapter 4 Exercises

All Prolog source code for Chap. 4 is available in the files `sieve.pl` and `draw.pl`. The LINUX shell scripts (S-4.1), p. 141, and (S-A.1), p. 195, are in the files `sieve` and `curves`, respectively.

Exercise 4.2. `circ_command/4` is defined in (P-A.10).

Prolog Code P-A.10: Definition of `circ_command/4` and Auxiliaries

```

1  circ(R, X, Y, Alpha, Pair) :-
2      Pi is 3.1415926,
3      Rad is Alpha * Pi / 180,
4      S is sin(Rad),
5      C is cos(Rad),
6      PairX is X + R * C,
7      PairY is Y + R * S,
8      concat_atom(['(',PairX,',',PairY,')'], Pair).

9  circ_pairs(R, X, Y, NInt, Pairs) :-
10     mesh(1, NInt, Mesh),
11     maplist(circ(R, X, Y), Mesh, Pairs).

12 circ_command(R, X, Y, NInt) :-
13     circ_pairs(R, X, Y, NInt, Pairs),
14     concat_atom(['\\newcommand{\\defcirc}{\\drawline'|Pairs], Atom),
15     concat_atom([Atom, '}', C),
16     write(C).

```

Illustration.

- ① A counterclockwise rotation by $\alpha = 60^\circ$ on a circle of radius $r = 10$ with centre at $(x, y) = (5, 2)$ maps the ‘rightmost’ point on the perimeter $(15, 2)$ to $(10, 10.6603)$.

```

?- circ(10, 5, 2, 0, P).
P = '(15,2)'
Yes
?- circ(10, 5, 2, 60, P).
P = '(10.0,10.6603)'
Yes

```

The output of `circ/5` is an atom.

- ② A uniformly spaced sequence of points on the circle’s perimeter is generated by `circ_pairs/5`. For example, points on the circle in ① spaced at $\alpha = 60^\circ (= 360^\circ/6)$, beginning with $(15, 2)$, are obtained by

```

?- circ_pairs(10, 5, 2, 6, Pairs).
Pairs = ['(15,2)', '(10.0,10.6603)', '(3.09401e-07,10.6603)',
        '(-5.0,2.0)', '(-6.18802e-07,-6.66025)', '(10.0,-6.66025)', '(15.0,2.0)']
Yes

```

circ_pairs/5 uses *mesh/3* ((P-4.4), p. 149) as an auxiliary. The output of *circ_pairs/5* is a list of atoms. They represent the co-ordinates of the points which will form the vertices of the approximating polygon. `\drawline` from *epic* will be used to connect them.

- ③ *circ_command/4* essentially concatenates the list entries from ② thus

```
?- circ_command(10, 5, 2, 6).
\newcommand{\defcirc}{\drawline(15,2)(10.0,10.6603)(3.09401e-07,10.6603)
(-5.0,2.0)(-6.18802e-07,-6.66025)(10.0,-6.66025)(15.0,2.0)}
Yes
```

- ④ The output from ③ is manually adjusted (in an editor) to result in the L^AT_EX definition

```
\newcommand{\defcirc}{\drawline(15,2)(10.0,10.6603)(3.09401e-07,10.6603)
(-5.0,2.0)(0,-6.66025)(10.0,-6.66025)(15.0,2.0)}
```

Exercise 4.3. The definition of *circ/5* is modified to *imp_circ/5* as shown in (P-A.11).

Prolog Code P-A.11: Definition of *imp_circ*/5

```

1  imp_circ(R, X, Y, Alpha, Pair) :-
2      Pi is 3.1415926,
3      Rad is Alpha * Pi / 180,
4      S is sin(Rad),
5      C is cos(Rad),
6      PairX is X + R * C,
7      sformat(SPairX, '~7f', PairX),
8      PairY is Y + R * S,
9      sformat(SPairY, '~7f', PairY),
10     concat_atom(['(',SPairX,',',',SPairY,')'], Pair).

```

Lines 6-9 in (P-A.11) illustrate the use of *sformat*/3; it unifies the value in floating point notation of a number with a string. Seven digits are used after the decimal point. The string then can serve as a component in the list of atoms in the first argument of *concat_atom*/2.

Rename *circ_pairs*/5 and *circ_command*/4 in (P-A.10) to *imp_circ_pairs*/5 and *imp_circ_command*/4, respectively, and also change in them all instances of *circ...* to *imp_circ...*. (These two predicates with these obvious changes are not shown here.)

Exercise 4.4. The definition of *gen_command2*/6 is shown in (P-A.12).

Prolog Code P-A.12: Definition of *gen_command2*/6

```

1  gen_mesh(Lower, Upper, NInt, Mesh) :-
2      Lower < Upper,
3      integer(NInt), NInt > 0,
4      gen_mesh(Lower, Upper, NInt, NInt, Mesh, []), !.

5  gen_mesh(Lower, _, _, 0, [Lower|Acc], Acc).
6  gen_mesh(Lower, Upper, NInt, NumInt, List, Acc) :-
7      H is Lower + NumInt * (Upper - Lower) / NInt,
8      NewNumInt is NumInt - 1,
9      gen_mesh(Lower, Upper, NInt, NewNumInt, List, [H|Acc]).

10 applic(Fun, Pars, Argument, Outcome) :- append(Pars, [Argument], List),
11                                         append(List, [Outcome], Args),
12                                         apply(Fun, Args).

13 gen_vals(Fun, Lower, Upper, NInt, Pars, Vals) :-
14     gen_mesh(Lower, Upper, NInt, Mesh),
15     maplist(applic(Fun, Pars), Mesh, Vals).

16 gen_command2(CName, Fun, Lower, Upper, NInt, Pars) :-
17     gen_vals(Fun, Lower, Upper, NInt, Pars, Vals),
18     concat_atom(['\\newcommand{', CName, '}{'\\drawline'|Vals}], Atom),
19     concat_atom([Atom, '}]'], Command),
20     write(Command).

```

gen_mesh/4 is defined by the accumulator technique using *gen_mesh*/6. In *applic*/4, first the argument list of *apply*/2 is assembled by list concatenation and then *apply*/2 is called. The remaining two predicates are

easily understood.

Exercise 4.5. The definition of *log_spiral/5* is shown in (P-A.13).

Prolog Code P-A.13: Definition of *log_spiral/5*

```

1 log_spiral(Alpha, CentreX, CentreY, RotAngle, Pair) :-
2   Pi is 3.1415926,
3   RadA is Alpha * Pi / 180,
4   SA is sin(RadA),
5   CA is cos(RadA),
6   K is CA/SA,
7   Phi is RotAngle * Pi / 180,
8   R is exp(K * Phi),
9   PairX is CentreX + R * cos(Phi),
10  sformat(SPairX, '~7f', PairX),
11  PairY is CentreY + R * sin(Phi),
12  sformat(SPairY, '~7f', PairY),
13  concat_atom(['(', SPairX, ',', SPairY, ')'], Pair).

```

Notice that the pattern set by (P A.11), p. 193, (the definition of the improved circle *imp_circ/5*) is broadly followed here. This applies in particular to the use of *sformat/3* for achieving a floating point representation of the points' co-ordinates. (As before, seven digits are used after the comma.)

Exercise 4.6. The definition of *curves/2* is shown in (P-A.14).

Prolog Code P-A.14: Definition of *curves/2*

```

1 curves(InFile, OutFile) :- see(InFile),
2                             tell(OutFile),
3                             execute,
4                             seen,
5                             told.
6
7 execute :- get_line(L),
8           ((L = ['\n'], execute);
9           (L = ['%'|_], copy_comment(L), execute);
10          (L = [end_of_file], true);
11          (exec_line(L), execute)).
12
13 copy_comment(List) :- atom_chars(Atom, List),
14                       write(Atom).
15
16 exec_line(Line) :- atom_chars(A, Line),
17                   term_to_atom(T, A),
18                   apply(T, []),
19                   write('\n').

```

Notice that the *execute/0* in (P-A.14) uses the predicate *get_line/1* defined in (P-4.2), p. 137. This predicate

reads from a file the next line as a *list* of characters.

Exercise 4.7. The definition of the shell script `curves` is shown in (S-A.1). It uses the temporary file `temp` for communicating the two filenames to the Prolog predicate `curves/2`. (This construct has been seen before in Sect. 4.1.4.)

LINUX Shell Script S-A.1: curves

```

1  #!/bin/bash
2  if [ $# -ne 2 ]; then
3      echo "Error: supply two arguments"
4  else
5      if [ -e $1 ]; then
6          echo $1 > temp
7          echo $2 >> temp
8      #
9          pl -f draw.pl -g go -t halt
10     #
11         echo "Input file : '$1'"
12         echo "Output file: '$2'"
13         echo "LaTeX source '$2' created"
14     #
15         rm temp
16     else
17         echo "Error: file '$1' does not exist"
18     fi
19 fi

```

It calls `go/0` (a predicate in `draw.pl`) which then uses `curves/2` from Exercise 4.6; `go/0` is defined in (P-A.15).

Prolog Code P-A.15: Definition of go/0

```

1  go :- see(temp),
2      get_string(InFile),
3      get_string(OutFile),
4      curves(InFile, OutFile).
5  %
6  % auxiliary predicate get_string/1 uses get_line/1 from (P-4.2), p. 137
7  %
8  get_string(String) :- get_line(List),
9                      append(ShortList, ['\n'],List),
10                     atom_chars(String, ShortList).

```

The auxiliary predicate `get_string/1` in (P-A.15) uses `get_line/1`, known from (P-4.2), p. 137.

Appendix B

Software

Below are listed all the filenames referenced in this book. The files are available on the Ventus website.

Referred to in Chap. 1

Prolog Source

enigma.pl

Referred to in Chap. 2

Prolog Source

automated.pl, bf.pl, bdf.pl, df.pl, df1.pl, df2.pl, df3.pl, df4.pl, blindsearches.pl, board.pl, eight_links.pl, eight_puzzle.pl, hand_knit.pl, iterd.pl, kinks.pl, kinks1.pl, kinks2.pl, kinks3.pl, kinks4.pl, kinks5.pl, links.pl, loop_puzzle1.pl, loop_puzzle1a.pl, loop_puzzle2.pl, loops.pl, naive.pl, netsearch.pl, searchinfo.pl, small_board.pl, straightloop.pl, straightloop1.pl, straightloop2.pl, straightloop3.pl

Referred to in Chap. 3

Prolog Source

asearches.pl, bsearches.pl, eight_puzzle_a.pl, eight_puzzle_b.pl, floorplan.pl, graph_a.pl, graph_b.pl, graph_c.pl, links.pl, knight.pl, maze.pl, maze_disp.pl, robot.pl, rsearches.pl, tedious.pl

Referred to in Chap. 4

Prolog Source

draw.pl, sieve.pl

L^AT_EX Source

exam.tex, part.tex, part_sln.tex, spirals.tex

Shell Script

sieve, curves

Other Files

spirals, without_waters, with_waters

References

- [1] K. Austin. Enigma 1225: Rows are columns. *New Scientist*, pages 55–55, 2003. February 8, 2003.
- [2] H.-J. Bartsch. *Handbook of Mathematical Formulas*. Academic Press, New York, 1974.
- [3] N. L. Biggs. *Discrete Mathematics*. Clarendon Press, Oxford, 1989.
- [4] H. Cambazard, B. O’Sullivan, and B.M. Smith. A constraint-based approach to enigma 1225. *Computers and Mathematics with Applications*, 58:1487–1497, 2009.
- [5] W. F. Clocksin. *Clause and Effect – Prolog Programming for the Working Programmer*. Springer, London, 1997.
- [6] M. A. Covington, D. Nute, and A. Vellino. *Prolog Programming in Depth*. Prentice Hall, Upper Saddle River, NJ, 1997.
- [7] A. Csenki. Enigma 1225: Prolog-assisted solution of a puzzle using discrete mathematics. *Computers and Mathematics with Applications*, 52:383–400, 2006.
- [8] A. Csenki. Rotations in the plane and Prolog. *Science of Computer Programming*, 66:154–161, 2007.
- [9] A. Csenki. *Prolog Techniques*. Ventus Publishing ApS, Copenhagen, 2009.
<http://www.bookboon.com/uk/student/it/>.
- [10] I. Fekete, T. Gregorics, and S. Nagy. *Bevezetés a Mesterséges Intelligenciába (Introduction to Artificial Intelligence)*. LSI Oktatóközpont a Mikroelektronika Kultúrájáért Alapítvány, Budapest, 1990.
- [11] M. Fogiel. *Handbook of Mathematical, Scientific, and Engineering Formulas, Tables, Graphs, Transforms*. Research and Education Association, New York, 1984.
- [12] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. CONSIT: a fully automated conditioned program slicer. *Software – Practice and Experience*, 34:15–46, 2004.
- [13] I. M. Gelfand and S. V. Fomin. *Calculus of Variations*. Prentice–Hall, Englewood Cliffs, NJ, 1963.
- [14] M. Goossens, F. Mittelbach, and A. Samarin. *The L^AT_EX Companion*. Addison–Wesley, Reading, Ma, 1994.
- [15] W. Jaksch. Künstliche Intelligenz I – Symbolische KI (Artificial Intelligence I – Symbolic AI). Technical report, University of Erlangen, Erlangen, Germany, 2002.
<http://www8.informatik.uni-erlangen.de/IMMD8/Lectures/KI-I/>.

-
- [16] EPS Trade Kft. Egyenes karika (Straight loop). *LOGIKOKTÉL, A Hungarian monthly puzzle magazine*, pages 2–2, 2001. Issue 2001/3.
 - [17] EPS Trade Kft. Fekete–Fehér (Black–White). *LOGIKOKTÉL, A Hungarian monthly puzzle magazine*, pages 2–2, 2001. Issue 2001/3.
 - [18] EPS Trade Kft. Minden második töréspont (Every other kink). *LOGIKOKTÉL, A Hungarian monthly puzzle magazine*, pages 10–10, 2002. Issue 2002/8.
 - [19] R. Knott. Using prolog to animate mathematics. In D. R. Brough, editor, *Logic Programming – New Frontiers*. Intellect Books, Oxford, 1992.
 - [20] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.
 - [21] R. E. Korf, M. Reids, and S. Edelkamp. Time complexity of iterative-deepening- A^* . *Artificial Intelligence*, 129:199–218, 2001.
 - [22] M. McGrath. *LINUX in Easy Steps*. Computer Step, Southam, 2006.
 - [23] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, San Francisco, Ca, 1998.
 - [24] D. S. Parker. Stream data analysis in prolog. In L. Shapiro, editor, *The Practice of Prolog*. MIT Press, Cambridge, Ma, 1990.
 - [25] H.-O. Peitgen, H. Jürgens, and D. Saupe. *Chaos and Fractals – New Frontiers of Science*. Springer, New York, 1992.
 - [26] K. F. Riley, M. P. Hobson, and S. J. Bence. *Mathematical Methods for Physics and Engineering*. Cambridge University Press, Cambridge, UK, second edition, 2002.
 - [27] S. J. Russell and P. Norvig. *Artificial Intelligence – A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.
 - [28] L. Sterling and E. Shapiro. *The Art of Prolog – Advanced Programming Techniques*. MIT Press, Cambridge Ma, London, 1986.
 - [29] T. Dean T, J. Allen, and Y. Aloimonos. *Artificial Intelligence – Theory and Practice*. Benjamin/Cummings, Redwood City Ca., 1995.
 - [30] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison–Wesley, Harlow and London and New York, 1996.
 - [31] S. Todd. *Basic Numerical Mathematics*, volume 2. Academic Press, Harlow and London and New York, 1978. Basic Numerical Algebra.
 - [32] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, 1984.
 - [33] J. Wielemaker. *SWI-Prolog 5.4 Reference Manual*. Amsterdam, 2004.
<http://www.wsi-prolog.org>.
 - [34] P. H. Winston. *Artificial Intelligence*. Addison–Wesley, Reading, Ma, third edition, 1992.
-

Index

- //, 165
- :, 49
- ==/2, 24
- !, 102
- \=/2, 24
- \==/2, 24
- acyclic graph, 128
- adjacency matrix, 110
- agenda, 54, 105, 108
- apply*/2, 155, 159
- atom_prefix*/2, 168
- call*/*n*, 40
- Cartesian product, 40
- city block distance, 110, 118, 130
- conduit model, 52
- consult*(*user*)
 - examples, 88, 108, 159, 184, 185
- cut, *see* !
- cycloid, 146–151
- dataflow diagram, 27
- derangement, 32
- difference lists, 67, 68
- discontiguous*, 181
- enumerator, *see* generator
- Ferrers Diagram, 36
- formatted output, 152
- functional programming, 27, 113
- generate-and-test, 17
- generator, 37–42
- get_char*/1, 138
- hand computations, 25, 27, 28
- Henderson diagram, *see* dataflow diagram
- heuristic, 103
 - admissible, 105, 114
 - alternative, 123–125
 - Euclidean, 123–124
 - zero, 123, 127
- heuristic evaluation function, 104
- higher order predicate, 40, 155
- int_to_atom*/2, 30
- interactive entry of code, *see* *consult*(*user*)
- keysort*/2, 108
- last*/2, 28
- L^AT_EX, 133–134, 143–160
- LINUX shell script, 139–145, 159, 195
- logarithmic spiral, 156, 194
- Manhattan distance, *see* city block distance
 - and the eight puzzle, 114
- maplist*/3, 150
 - and functional programming, 27
- memoization, 120
- Minkowski Inequality, 132
- mod*, 166
- module*/2, *see* modules
- modules, 47–49
- partial application, 150, 155, 172
- partition of a number
 - definition of, 33
 - generating partitions, 35–36
- pattern matching, 139
- problems for Prolog
 - L^AT_EX code generation, 146–151
 - drawing with L^AT_EX, 146–160
 - eight puzzle, 99–102, 114–118

- knight, 128–132
 - loop puzzles, 76–96
 - maze, 121–128
 - robot navigation, 118–120
 - Rows are Columns, 17–46
 - text removal, 133–145
 - text retention, 151
- relaxed problem, 114
- rotation
 - list rotation, 43
 - rotation of a cycle, 32
- search, 47–128
 - blind search, 47–102
 - Bounded Depth First, 68–72
 - Breadth First, 67–68
 - Depth First, 52–67
 - Iterative Deepening, 72–74
 - informed search, 103–128
 - A-Algorithm, 105–108
 - Best First, 118
 - Hill Climbing, 118
 - Iterative Deepening A^* , 108–110
 - Iterative Deepening $A^*-\epsilon$, 109
- search tree, 49
- see/1*, 138
- seen/0*, 138
- sformat/3*, 152, 193, 194
- shell script, *see* LINUX shell script
- slicing, 133
- snd/2*
 - and functional programming, 27
 - definition of, 25
- sort/2*, 25
- stream data analysis, 27
- tail recursion, 102
- text processing, 133–160
- Triangle Inequality, 131
- unify_with_occurs_check/2*, 181
- use_module/1*, *see* modules
- writeln/2*, 30
- zip/3*
- and functional programming, 27
- definition of, 163
- specification of, 25

Errata to Volume 1

Correct the following typesetting errors in [9].

- Page 133: remove the fourth line of the second verse, i.e. the line ‘Went to mow a meadow,’.
- Page 183: replace in reference [14] ‘N. J. Nilsson’ by ‘S. J. Russell’.

The author welcomes comments and observations on his Prolog books published by Ventus.