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_ilist(+Width,+List)* by

```
write_ilist(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).
```

$$\text{total([(1,10),(2,100),(3,1000)],Total)} \overset{⓪}{\rightsquigarrow}$$
$$\text{total([(1,10),(2,100),(3,1000)],0,Total)} \overset{②}{\rightsquigarrow}$$
$$\text{total([(2,100),(3,1000)],10,Total)} \overset{②}{\rightsquigarrow}$$
$$\text{total([(3,1000)],210,Total)} \overset{②}{\rightsquigarrow} \text{total([],3210,Total)} \overset{①}{\rightsquigarrow}$$
$$\text{Total} = 3210 \overset{⓪}{\rightsquigarrow} \text{success}$$

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_ilist(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_imatrix(Matrix) :- largest(Matrix,Max),
                         ndigits(Max,ND),
                         Width is ND + 2,
                         write_imatrix(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_imatrix/2` with

  ```
  write_imatrix(_,[]).
  write_imatrix(Width, [H|T]) :- write_ilist(Width, H), nl,
                                 write_imatrix(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.

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

Current Partition	$[1^5 5^1 6^2]$	$[1^3 5^1 7^2]$	$[1^5 4^3 5^1]$
Next Partition	$[2^1 4^2 6^2]$	$[4^2 7^2]$	$[3^3 4^2 5^1]$
Step Used	(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

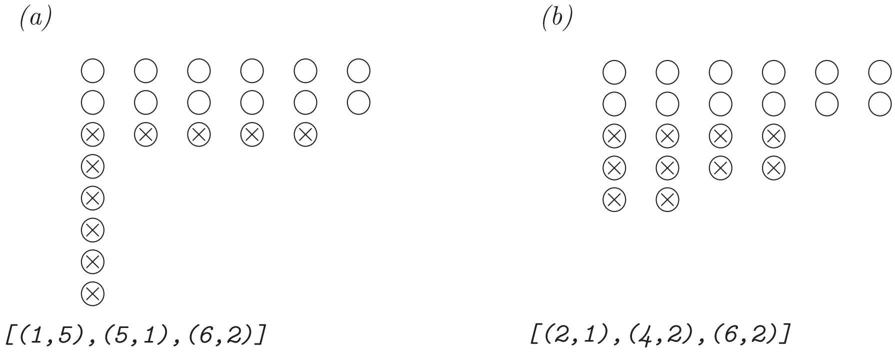


Figure A.2: Ferrers Diagrams and their Prolog Representations

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).

---

[1] We have met *atom_concat/3* in [9, p. 138].

$$
\begin{array}{l}
\mathit{split}([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16],\ [(2,1),(3,3),(5,1)],\ [],\ S)\ \overset{\textcircled{3}}{\rightsquigarrow}\\
\mathit{split}([3,4,5,6,7,8,9,10,11,12,13,14,15,16],\ [(2,0),(3,3),(5,1)],\ [[1,2]],\ S)\ \overset{\textcircled{2}}{\rightsquigarrow}\\
\mathit{split}([3,4,5,6,7,8,9,10,11,12,13,14,15,16],\ [(3,3),(5,1)],\ [[1,2]],\ S)\ \overset{\textcircled{3}}{\rightsquigarrow}\\
\mathit{split}([6,7,8,9,10,11,12,13,14,15,16],\ [(3,2),(5,1)],\ [[3,4,5],\ [1,2]],\ S)\ \overset{\textcircled{3}}{\rightsquigarrow}\\
\mathit{split}([9,10,11,12,13,14,15,16],\ [(3,1),(5,1)],\ [[6,7,8],\ [3,4,5],\ [1,2]],\ S)\ \overset{\textcircled{3}}{\rightsquigarrow}\\
\mathit{split}([12,13,14,15,16],\ [(3,0),(5,1)],\ [[9,10,11],\ [6,7,8],\ [3,4,5],\ [1,2]],\ S)\ \overset{\textcircled{2}}{\rightsquigarrow}\\
\mathit{split}([12,13,14,15,16],\ [(5,1)],\ [[9,10,11],\ [6,7,8],\ [3,4,5],\ [1,2]],\ S)\ \overset{\textcircled{3}}{\rightsquigarrow}\\
\mathit{split}([],\ [(5,0)],\ [[12,13,14,15,16],\ [9,10,11],\ [6,7,8],\ [3,4,5],\ [1,2]],\ S)\ \overset{\textcircled{1}}{\rightsquigarrow}\\
\mathit{reverse}([[12,13,14,15,16],\ [9,10,11],\ [6,7,8],\ [3,4,5],\ [1,2]],\ S)\ \rightsquigarrow\\
S = [[1,2],\ [3,4,5],\ [6,7,8],\ [9,10,11],\ [12,13,14,15,16]]\ \rightsquigarrow\ \mathit{success}
\end{array}
$$

Figure A.4: Annotated Hand Computations for *split/4*

Prolog Code P-A.1: *Definition of* `split/4`

```
split([],[(_,0)],Acc,S)       :- reverse(Acc,S), !. % clause 1
split(L,[(_,0)|T],Acc,S)      :- split(L,T,Acc,S).  % clause 2
split(L,[(K,AlphaK)|T],Acc,S) :-                    % clause 3
   AlphaK > 0,                                      %
   append(L1,L2,L),                                 %
   length(L1,K),                                    %
   NewAlphaK is AlphaK - 1,                         %
   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).
```

depth_first(d,c) $\overset{\textcircled{0}}{\rightsquigarrow}$ dfs_loop([d],[],c) $\overset{\textcircled{2}}{\rightsquigarrow}$

dfs_loop([e,s,a],[d],c) $\overset{\textcircled{2}}{\rightsquigarrow}$ dfs_loop([f,b,s,a],[e,d],c) $\overset{\textcircled{2}}{\rightsquigarrow}$

dfs_loop([g,b,s,a],[f,e,d],c) $\overset{\textcircled{2}}{\rightsquigarrow}$ dfs_loop([b,s,a],[g,f,e,d],c) $\overset{\textcircled{2}}{\rightsquigarrow}$

dfs_loop([c,a,s,a],[b,g,f,e,d],c) $\overset{\textcircled{1}}{\rightsquigarrow}$ success

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*

```
extend_path(Nodes,Path,ExtendedPath) :-
   maplist(glue(Path),Nodes,ExtendedPath).

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*

```
extend_path([],_,[]).                                      % clause 1
extend_path([Node|Nodes],Path,[[Node|Path]|Extended]) :-   % clause 2
   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: [f]
Open: [b, s, a], Closed: [g, f, e, d]
Node b is being expanded. Successors: [c, e, a]
Open: [c, a, s, a], Closed: [b, g, f, e, d]
Goal found: c
Yes
```

Figure A.6: Interactive Session for the Query `?- depth_first(d,c).`

$$\mathit{depth\_first}(u,c) \overset{\textcircled{0}}{\rightsquigarrow} \mathit{dfs\_loop}([u],[],c) \overset{\textcircled{2}}{\rightsquigarrow}$$
$$\mathit{dfs\_loop}([v,w],[u],c) \overset{\textcircled{2}}{\rightsquigarrow} \mathit{dfs\_loop}([w,w],[v,u],c) \overset{\textcircled{2}}{\rightsquigarrow}$$
$$\mathit{dfs\_loop}([w],[w,v,u],c) \overset{\textcircled{2}}{\rightsquigarrow} \mathit{dfs\_loop}([],[w,w,v,u],c) \rightsquigarrow \mathit{failure}$$

Figure A.7: Hand Computations for the Query `?- depth_first(u,c).`

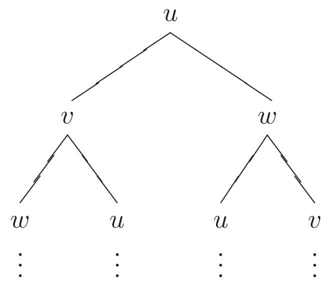


Figure A.8: Tree for Finding Successor Nodes in the New Component

```
?- 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.

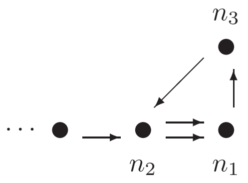
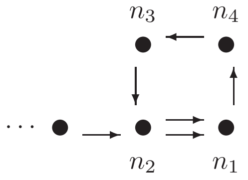
Example Path	*Prolog Clause*
	`same([N1,N2,N3,N1,N2|_]).`
	`same([N1,N2,N3,N4,N1,N2|_]).`

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
```

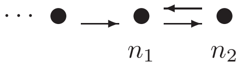
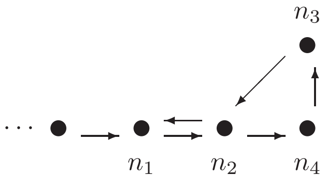
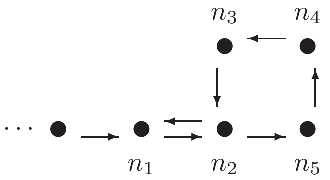
<table>
<tr><th>Example Path</th><th>Prolog Clause</th></tr>
<tr><td> </td><td>opposite([N1,N2,N1|_]).</td></tr>
<tr><td> </td><td>opposite([N1,N2,N3,N4,N2,N1|_]).</td></tr>
<tr><td> </td><td>opposite([N1,N2,N3,N4,N5,N2,N1|_]).</td></tr>
</table>

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).
```

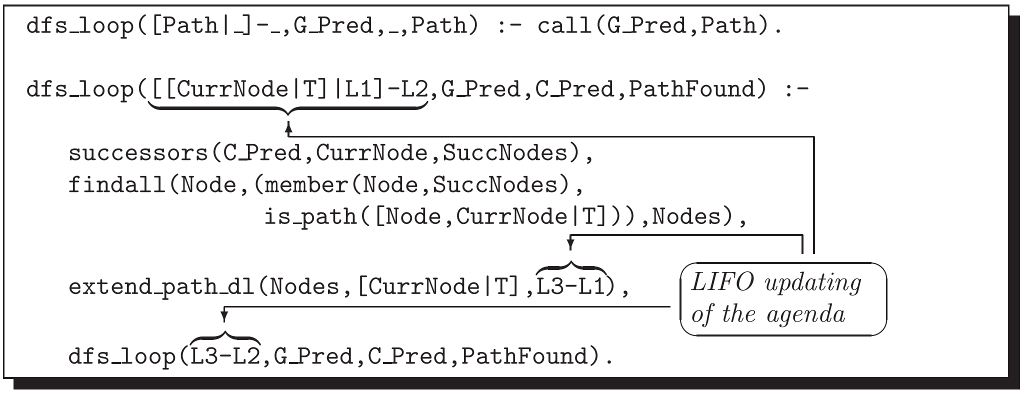
Figure A.11: Definition of *extend_path_dl/3*

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*

```
depth_first_dl(Start,G_Pred,C_Pred,PathFound) :-
   dfs_loop([[Start]|L]-L,G_Pred,C_Pred,PathFoundRev),
   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_dl(Nodes,[CurrNode|T],L3-L1),

   dfs_loop(L3-L2,G_Pred,C_Pred,PathFound).
```

Figure A.12: New Clauses for *dfs_loop/4*

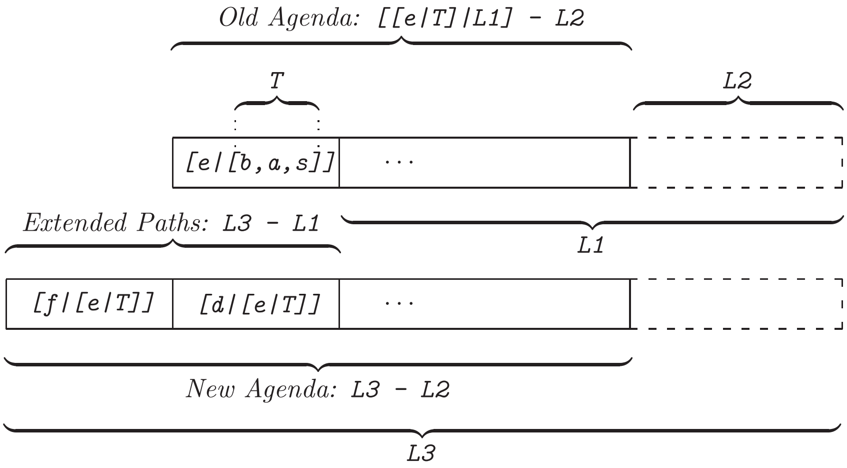


Figure A.13: Updating of the Agenda by *dfs_loop/4*

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
```

---

[2] In 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(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`

```
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-file[3]
% 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
```

---

[3] To suppress this warning message, place the directive

`:- discontiguous dfs_loop/4.`

just after the *use_module* directives in `df.pl`.

```
:- discontiguous 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).
```

FIFO updating of the agenda

Copied from the augmented version of `df.pl` *(Exercise 2.5, Fig. A.11, p. 177)*

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.

```
b_dfs_loop([Path|_],G_Pred,_,_,Path) :- call(G_Pred,Path).
b_dfs_loop([[CurrNode|T]|Others],G_Pred,C_Pred,Hor,PathFound) :-
   length([CurrNode|T],ListLength),
   PathLength is ListLength - 1,
   PathLength < Hor,
   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),
   b_dfs_loop(NewOpenPaths,G_Pred,C_Pred,Hor,PathFound).
b_dfs_loop([[CurrNode|T]|Others],G_Pred,C_Pred,Hor,PathFound) :-
   length([CurrNode|T],ListLength),
   PathLength is ListLength - 1,
   PathLength >= Hor,
   b_dfs_loop(Others,G_Pred,C_Pred,Hor,PathFound).
```

Clause essentially as in Fig. 2.15, p. 65

New goals due to the presence of the horizon

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`

```
b_dfs_loop([Path|_],G_Pred,_,_,Path) :- call(G_Pred,Path),
                                        lastdepth(LastDepth),
                                        length(Path,ListLength),
                                        PathLength is ListLength - 1,
                                        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*

---

[4] Here 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`

```
goal_path([H|T]) :- number_of_marks(M),
                    length([H|T],M),
                    last(E,T),
                    link(H,E),
                    size(Row,Col),       % added goal
                    Size is Row * Col,   % added goal
                    flatten([H|T],F),    % added goal
                    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`

```
e_cost(Node,Goal,D) :- in(Node,X1,Y1),
                       in(Goal,X2,Y2),
                       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* `e_cost/3`

```
edge_cost(Node1,Node2,Cost) :- link(Node1,Node2),
                               in(Node1,X1,Y1),
                               in(Node2,X2,Y2),
                               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] $\overset{\textcircled{1}}{\rightsquigarrow}$

[8.24-[2,1]-4, 11.83-[3,1]-6] $\overset{\textcircled{2}}{\rightsquigarrow}$

[8.24-[2,1]-4, 11.83-[3,1]-6] $\overset{\textcircled{1}}{\rightsquigarrow}$

[10-[4,2,1]-8, 9.24-[5,2,1]-7, 12.39-[6,2,1]-7, 11.83-[3,1]-6] $\overset{\textcircled{2}}{\rightsquigarrow}$

[9.24-[5,2,1]-7, 10-[4,2,1]-8, 11.83-[3,1]-6, 12.39-[6,2,1]-7] $\overset{\textcircled{1}}{\rightsquigarrow}$

[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] $\overset{\textcircled{2}}{\rightsquigarrow}$

[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] $\overset{\textcircled{1}}{\rightsquigarrow}$

[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] $\overset{\textcircled{2}}{\rightsquigarrow}$

[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] $\overset{\textcircled{3}}{\rightsquigarrow}$ *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 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.)

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)

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.

---

[5] The 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*

```
circ(R, X, Y, Alpha, Pair) :-
   Pi is 3.1415926,
   Rad is Alpha * Pi / 180,
   S is sin(Rad),
   C is cos(Rad),
   PairX is X + R * C,
   PairY is Y + R * S,
   concat_atom(['(',PairX,',',PairY,')'], Pair).

circ_pairs(R, X, Y, NInt, Pairs) :-
   mesh(1, NInt, Mesh),
   maplist(circ(R, X, Y), Mesh, Pairs).

circ_command(R, X, Y, NInt) :-
   circ_pairs(R, X, Y, NInt, Pairs),
   concat_atom(['\\newcommand{\\defcirc}{\\drawline'|Pairs], Atom),
   concat_atom([Atom,'}'], C),
   write(C).
```

Illustration.

① A counterclockwise rotation by $\alpha = 60°$ 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 $\alpha = 60°(= 360°/6)$, 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 LaTeX 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`

```
gen_mesh(Lower, Upper, NInt, Mesh) :-
   Lower < Upper,
   integer(NInt), NInt > 0,
   gen_mesh(Lower, Upper, NInt, NInt, Mesh, []), !.

gen_mesh(Lower, _, _, 0, [Lower|Acc], Acc).
gen_mesh(Lower, Upper, NInt, NumInt, List, Acc) :-
   H is Lower + NumInt * (Upper - Lower) / NInt,
   NewNumInt is NumInt - 1,
   gen_mesh(Lower, Upper, NInt, NewNumInt, List, [H|Acc]).

applic(Fun, Pars, Argument, Outcome) :- append(Pars, [Argument], List),
                                        append(List, [Outcome], Args),
                                        apply(Fun, Args).

gen_vals(Fun, Lower, Upper, NInt, Pars, Vals) :-
   gen_mesh(Lower, Upper, NInt, Mesh),
   maplist(applic(Fun, Pars), Mesh, Vals).

gen_command2(CName, Fun, Lower, Upper, NInt, Pars) :-
   gen_vals(Fun, Lower, Upper, NInt, Pars, Vals),
   concat_atom([’\\newcommand{’, CName, ’}{\\drawline’|Vals], Atom),
   concat_atom([Atom,’}’], Command),
   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`

```
log_spiral(Alpha, CentreX, CentreY, RotAngle, Pair) :-
   Pi is 3.1415926,
   RadA is Alpha * Pi / 180,
   SA is sin(RadA),
   CA is cos(RadA),
   K is CA/SA,
   Phi is RotAngle * Pi / 180,
   R is exp(K * Phi),
   PairX is CentreX + R * cos(Phi),
   sformat(SPairX, '~7f',PairX),
   PairY is CentreY + R * sin(Phi),
   sformat(SPairY, '~7f',PairY),
   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`

```
curves(InFile, OutFile) :- see(InFile),
                           tell(OutFile),
                           execute,
                           seen,
                           told.

execute :- get_line(L),
           ((L = ['\n'], execute);
            (L = ['%'|_], copy_comment(L), execute);
            (L = [end_of_file], true);
            (exec_line(L), execute)).

copy_comment(List) :- atom_chars(Atom,List),
                      write(Atom).

exec_line(Line) :- atom_chars(A,Line),
                   term_to_atom(T,A),
                   apply(T,[]),
                   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`

```
#!/bin/bash
if [ $# -ne 2 ]; then
   echo "Error: supply two arguments"
else
   if [ -e $1 ]; then
      echo $1 > temp
      echo $2 >> temp
#
      pl -f draw.pl -g go -t halt
#
      echo "Input file : ’$1’"
      echo "Output file: ’$2’"
      echo "LaTeX source ’$2’ created"
#
      rm temp
   else
      echo "Error: file ’$1’ does not exist"
   fi
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*

```
go :- see(temp),
      get_string(InFile),
      get_string(OutFile),
      curves(InFile, OutFile).
%
% auxiliary predicate get_string/1 uses get_line/1 from (P-4.2), p. 137
%

get_string(String) :- get_line(List),
                      append(ShortList, [’\n’],List),
                      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.