

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.

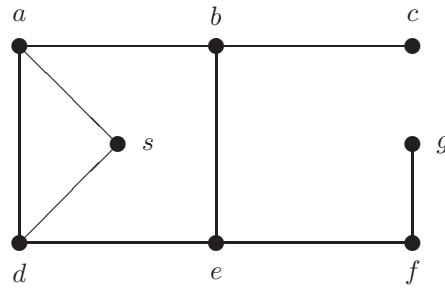


Figure 2.1: A Network

```

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

```

I joined MITAS because
I wanted **real responsibility**

The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com

Month 16

I was a construction supervisor in the North Sea advising and helping foremen solve problems

Real work
International opportunities
Three work placements

```

:- 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 ;
...

```

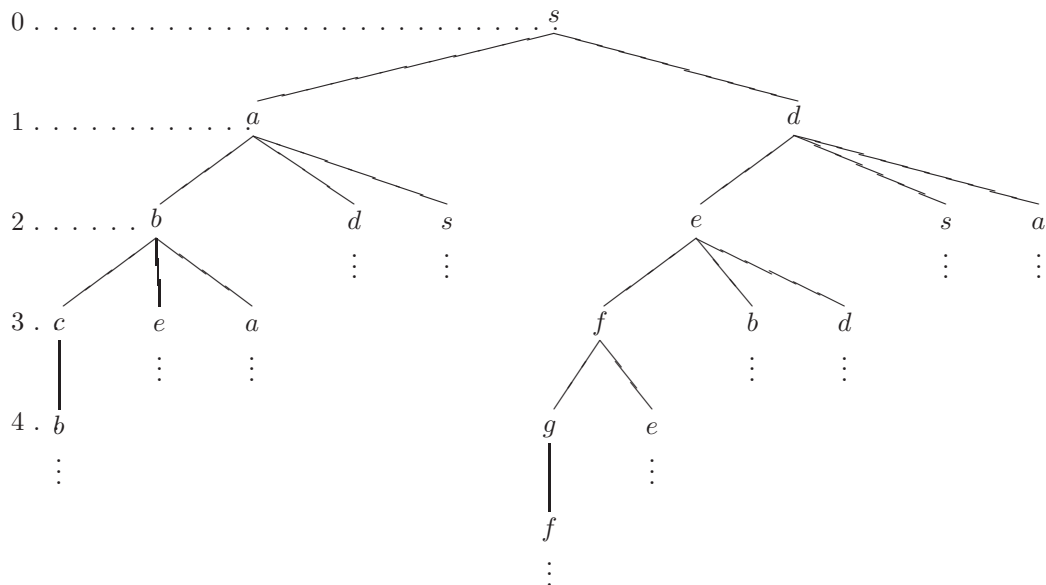


Figure 2.4: The Search Tree

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).
```

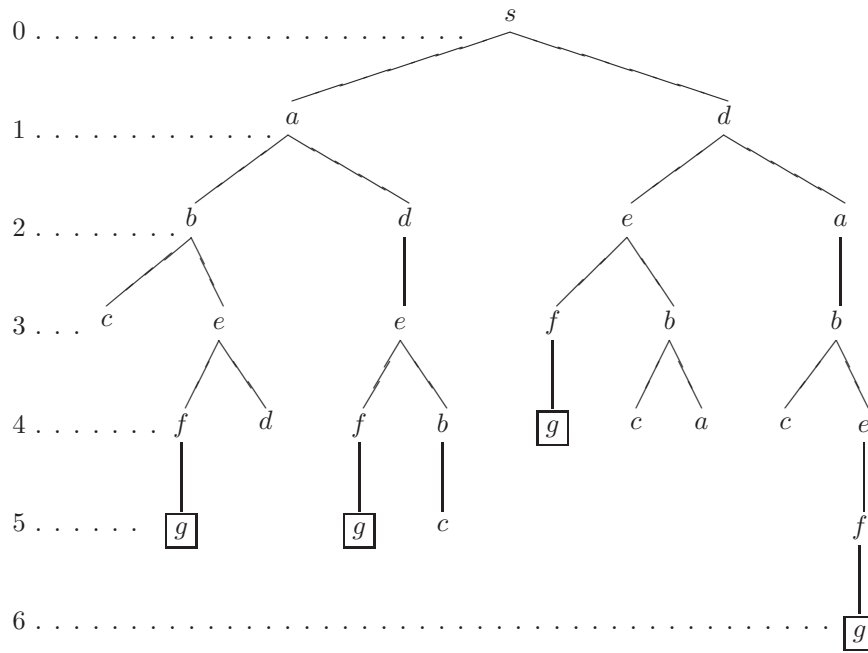


Figure 2.5: The Pruned Search Tree

ie business school

#1 EUROPEAN BUSINESS SCHOOL
FINANCIAL TIMES 2013

#gobeyond

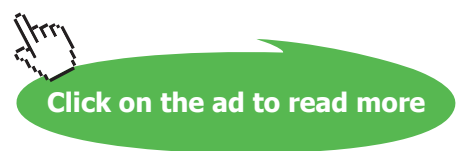
MASTER IN MANAGEMENT

Because achieving your dreams is your greatest challenge. IE Business School's Master in Management taught in English, Spanish or bilingually, trains young high performance professionals at the beginning of their career through an innovative and stimulating program that will help them reach their full potential.

- Choose your area of specialization.
- Customize your master through the different options offered.
- Global Immersion Weeks in locations such as London, Silicon Valley or Shanghai.

Because you change, we change with you.

www.ie.edu/master-management | mim.admissions@ie.edu | f t in YouTube



```
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.

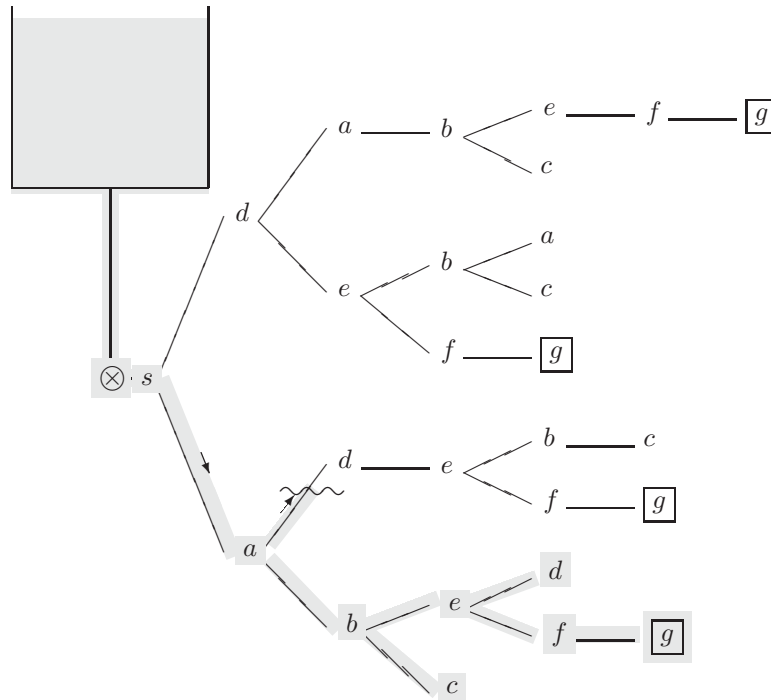


Figure 2.6: Depth First Search – The Conduit Model

“I studied English for 16 years but...
...I finally learned to speak it in just six lessons”
Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



```

:- 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*

	{	<i>SuccList</i> \leftarrow successors of <i>H</i>
		<i>OpenList</i> \leftarrow <i>SuccList</i> ++ <i>T</i>
do	{	if <i>OpenList</i> = []
		then return (<i>failure</i>)
		<i>H T</i> \leftarrow <i>OpenList</i>

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:



**university of
 groningen**

Excellent Economics and Business programmes at:





**“The perfect start
 of a successful,
 international career.”**

www.rug.nl/feb/education

CLICK HERE
 to discover why both socially
 and academically the University
 of Groningen is one of the best
 places for a student to be



```

:- 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 ≠ GoalNode
do {
  SuccList ← successors of H (1)
  OpenList ← (SuccList ∩ ClosedListc) ++ T (2)
  ClosedList ← [H|ClosedList] (3)
  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), %
                not(member(Node,ClosedList))),Nodes), } ← Implements (2)
  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

LIGS University

based in Hawaii, USA

is currently enrolling in the
Interactive Online **BBA, MBA, MSc,**
DBA and PhD programs:

- ▶ enroll **by October 31st, 2014** and
- ▶ **save up to 11%** on the tuition!
- ▶ pay in 10 installments / 2 years
- ▶ Interactive **Online education**
- ▶ visit www.ligsuniversity.com to
find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education. More info [here](#).



```

?- 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

```

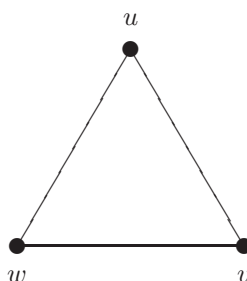
Figure 2.11: Illustrative Query for *depth_first/2* – Second Version

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*.

.....Alcatel-Lucent 

www.alcatel-lucent.com/careers

What if you could build your future and create the future?

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".



```

depth_first(s, g, Path) ~>
dfs_loop([[s]], [], g, Path) ~>
dfs_loop([[a,s], [d,s]], [s], g, Path) ~>
dfs_loop([[b,a,s], [d,a,s], [g,d,s], [d,s]], [a,s], g, Path) ~>
dfs_loop([[c,b,a,s], [e,b,a,s], [g,b,d,s], [d,a,s], [d,s]], [b,a,s], g, Path) ~>
dfs_loop([[b,d,b,d,b,d], [e,b,a,s], [d,a,s], [d,s]], [c,b,a,s], g, Path) ~>
dfs_loop([[f,e,b,a,s], [b,d,b,d,b,d], [d,e,b,a,s], [d,a,s], [d,s]], [e,c,b,a,s], g, Path) ~>
dfs_loop([[g,f,e,b,a,s], [e,f,e,f,e,f], [d,e,b,a,s], [d,a,s], [d,s]], [f,e,c,b,a,s], g, Path) ~>
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]) ~>
depth_first(s, g, [g,f,e,b,a,s]) ~> 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  $H \neq GoalNode$ 
    do  $\left\{ \begin{array}{l} SuccList \leftarrow \text{successors of } H \\ NewOpenNodes \leftarrow (SuccList \cap ClosedList^c) \\ NewPaths \leftarrow EXTENDPATH(NewOpenNodes, [H|T]) \\ OpenPaths \leftarrow NewPaths ++ TailOpenPaths \\ \text{if } OpenPaths = [] \\ \quad \text{then return } (failure) \\ [[H|T]|TailOpenPaths] \leftarrow OpenPaths \end{array} \right.$ 
  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



Maastricht University *Leading in Learning!*

Join the best at the Maastricht University School of Business and Economics!

Top master's programmes

- 33rd place Financial Times worldwide ranking: MSc International Business
- 1st place: MSc International Business
- 1st place: MSc Financial Economics
- 2nd place: MSc Management of Learning
- 2nd place: MSc Economics
- 2nd place: MSc Econometrics and Operations Research
- 2nd place: MSc Global Supply Chain Management and Change

Sources: Keuzegids Master ranking 2013; Elsevier 'Beste Studies' ranking 2012; Financial Times Global Masters in Management ranking 2012

Visit us and find out why we are the best!
Master's Open Day: 22 February 2014

Maastricht University is the best specialist university in the Netherlands (Elsevier)

www.mastersopenday.nl

- 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

$\left\{ \begin{array}{l} \textit{SuccList} \leftarrow \text{successors of } H \\ \textit{ONodes} \leftarrow \text{list of } S \in \textit{SuccList} \text{ with } \text{ISPATH}([S, H|T]) \\ \textit{NewPaths} \leftarrow \text{EXTENDPATH}(\textit{ONodes}, [H|T]) \\ \textit{OpenPaths} \leftarrow \textit{NewPaths} \text{ ++ } \textit{TailOpenPaths} \end{array} \right.$

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

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.



agency: cdfg - © Photonstop

> Apply now

REDEFINE YOUR FUTURE
**AXA GLOBAL GRADUATE
PROGRAM 2015**

redefining / standards 

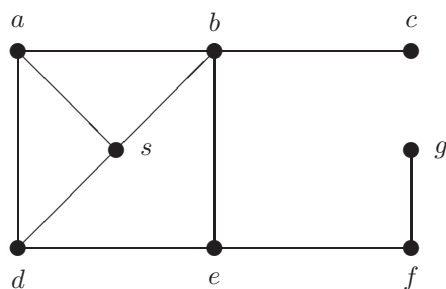


Figure 2.18: A Network (see Exercise 2.4, p. 67)

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:

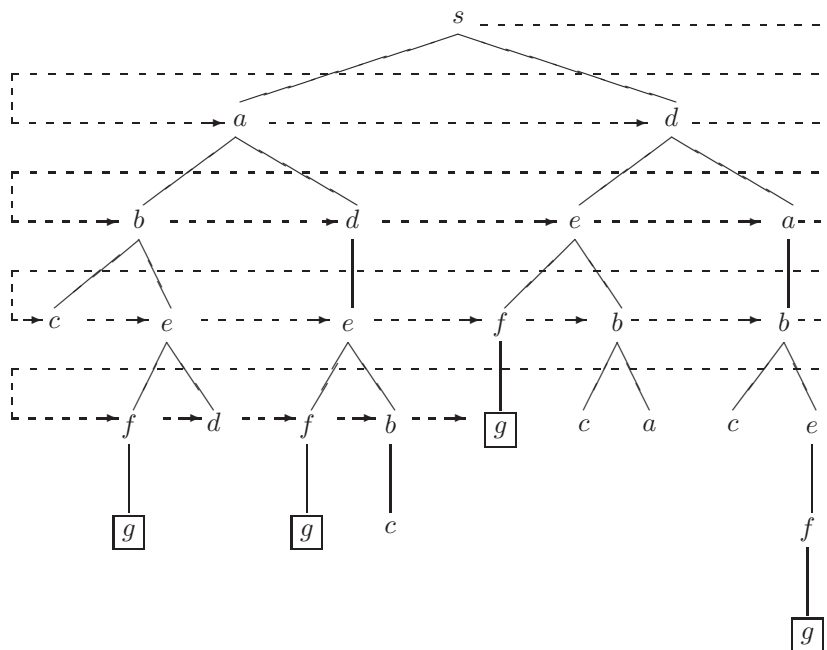


Figure 2.19: Breadth First

- 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.



Empowering People. Improving Business.

BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

www.bi.edu/master

BI NORWEGIAN BUSINESS SCHOOL

EFMD EQUIS ACCREDITED



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 ...
...

```

Loop Predicate *b_dfs_loop/5*
to be defined here

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
```

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now



Go to www.helpmyassignment.co.uk for more info

 **Helpmyassignment**



```

?- 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
Ctrl+C after this
Action (h for help) ? abort
% Execution Aborted
?- lastdepth(D).
D = 396 } ← Last value of
Yes horizon

```

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].



Brain power

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations. Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

SKF



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 |
| | | | | | | |
+---+---+---+---+---+

Stop search? (y/n) y.
Yes

```

Figure 2.27: Sample Session – The Loop Puzzle


```

:- use_module(loops).      } ← Top level module is in loops.pl
size(6,6).                } ← Number of rows
                           } ← Number of columns
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 2.28: 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.

“I studied English for 16 years but...
...I finally learned to speak it in just six lessons”
Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download

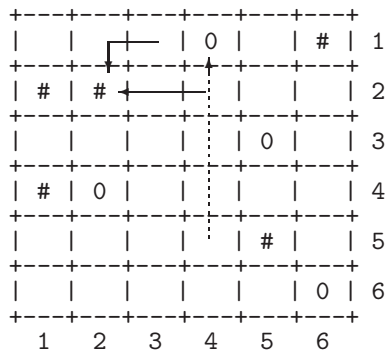


Figure 2.29: Constructing a Solution of the Loop Puzzle

(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

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

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).

... } ← Define link/2 here (see Exercise 2.11, p. 81)

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).
                   } ← The goal path has 9 segments
                   } ← The goal path is closed

is_path([H|T]) :- not(prohibit([H|T])).
                   } ← Exclude self-intersecting paths

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: 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)],

```

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

¹²A 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)]
```



What do you want to do?

No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site www.volvogroup.com. We look forward to getting to know you!

VOLVO
 AB Volvo (publ)
www.volvogroup.com

VOLVO TRUCKS | RENAULT TRUCKS | MACK TRUCKS | VOLVO BUSES | VOLVO CONSTRUCTION EQUIPMENT | VOLVO PENTA | VOLVO AERO | VOLVO IT
 VOLVO FINANCIAL SERVICES | VOLVO 3P | VOLVO POWERTRAIN | VOLVO PARTS | VOLVO TECHNOLOGY | VOLVO LOGISTICS | BUSINESS AREA ASIA

```

:- 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]).
                                     } ← For Exercise 2.13 only
...
} ← Define link/2 here (see Exercise 2.12)
...
} ← Define maybe_start_state/1 here (see Exercise 2.13)
goal_path([H|T]) :- number_of_marks(M),
                    length([H|T],M),
                    last(E,T),
                    link(H,E).
                                     } ← Modified definition
                                     } ← of goal_path/1
                                     } ← (see Exercise 2.13)
...
} ← Define number_of_marks/1 here (see Exercise 2.13)
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).
                                     } ← Copy from
                                     } ← hand_knit.pl
                                     } ← (see Fig. 2.30)

```

Figure 2.32: The File automated.pl

gaiTEYE
Challenge the way we run

**EXPERIENCE THE POWER OF
FULL ENGAGEMENT...**

**RUN FASTER.
RUN LONGER..
RUN EASIER...**

**READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM**


```
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

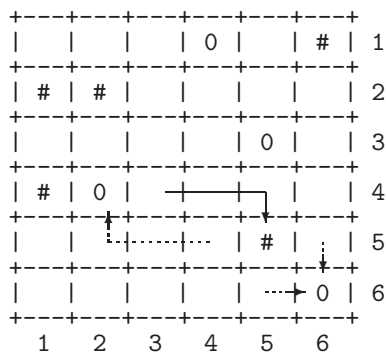


Figure 2.33: Constructing a Loop

- *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.

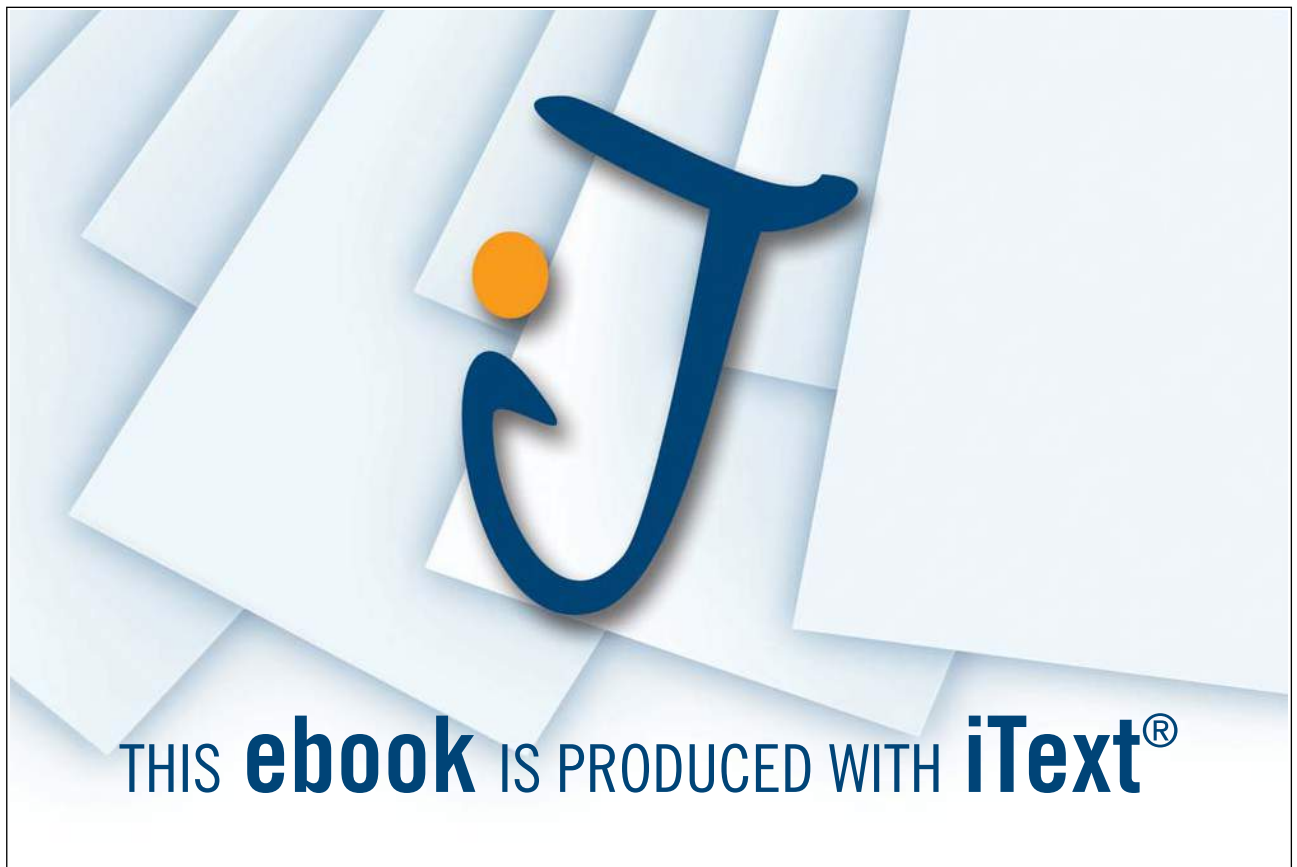
■

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



www.sylvania.com

We do not reinvent
the wheel we reinvent
light.

Fascinating lighting offers an infinite spectrum of possibilities: Innovative technologies and new markets provide both opportunities and challenges. An environment in which your expertise is in high demand. Enjoy the supportive working atmosphere within our global group and benefit from international career paths. Implement sustainable ideas in close cooperation with other specialists and contribute to influencing our future. Come and join us in reinventing light every day.

Light is OSRAM

OSRAM
SYLVANIA




```
?- 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.

```

?- consult([loop_puzzle1a, board]).
% blindsearches compiled into blindsearches 0.05 sec, 7,380 bytes
% automated compiled into auto 0.00 sec, 6,252 bytes
% loop_puzzle1a compiled 0.05 sec, 15,076 bytes
% board compiled into board 0.06 sec, 8,168 bytes
?- maybe_start_state(_Start),
   depth_first(_Start,goal_path,link,_PathFound),
   reverse(_PathFound,_Rev), flatten(_Rev,_F), last(_L,_F),
   size(_Row,_Col), make_board(_Row,_Col,[_L|_F],_B0),
   bagof(_C,circle(_C),_Cs), change_board('0',_Cs,_B0,_B1),
   bagof(_S,sharp(_S),_Ss), change_board('#',_Ss,_B1,_B2),
   show_board(_B2).
+-----+-----+-----+-----+-----+
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
+-----+-----+-----+-----+-----+
| * | * | * | * | * | * | * |
| # | # | # | # | # | # | # |
| * | * | * | * | * | * | * |
+-----+-----+-----+-----+-----+
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
+-----+-----+-----+-----+-----+
| * | * | * | * | * | * | * |
| # | # | # | # | # | # | # |
| * | * | * | * | * | * | * |
+-----+-----+-----+-----+-----+
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
+-----+-----+-----+-----+-----+
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
+-----+-----+-----+-----+-----+
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
+-----+-----+-----+-----+-----+
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
| * | * | * | * | * | * | * |
+-----+-----+-----+-----+-----+

```

Yes

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

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

Figure 2.40: Illustrating Exercise 2.19

- You may find that due to stack overflow your Prolog implementation won't be able to solve this more complex puzzle by Breadth First because the agenda will become very large (Sect. 2.5).

■

2.8.7 Avoiding Multiple Solutions

This may be another desired feature of the implementation: Every loop satisfying the specifications should be displayed only once. There are two ways a solution may be discovered more than once.

- As loops can be traversed in two directions, both versions will be found even though the display won't allow us to distinguish between them. To illustrate the point, let us consider the loop shown in Fig. 2.42. We take `pos(2,7)` to be the seed position. Then the loop can be built up by starting with the segment

$$[\text{pos}(5,7), \text{pos}(4,7), \text{pos}(3,7)]$$

bearing in mind that segments are read from right to left. Alternatively,

$$[\text{pos}(1,1), \text{pos}(2,1), \text{pos}(2,2), \text{pos}(2,3), \text{pos}(2,4), \text{pos}(2,5), \text{pos}(2,6)]$$

may also be taken as the starting segment emanating from the same seed. It starts the loop in the opposite direction. We won't be concerned here with duplication due to this cause; we simply accept that *as far as this cause is concerned* each solution of the puzzle will be displayed exactly twice.

- The second cause for finding multiple instances of the same loop is elusive and it won't arise with every test case. The case shown in Fig. 2.42 is, however, one of those where this will occur. One of the segments emanating from the seed position `pos(2,7)` is `[pos(1,6), pos(1,7)]`, pointing to the sharp in `pos(1,6)`. The same segment can also be thought of, however, as emanating from the sharp in `pos(1,8)`. This is

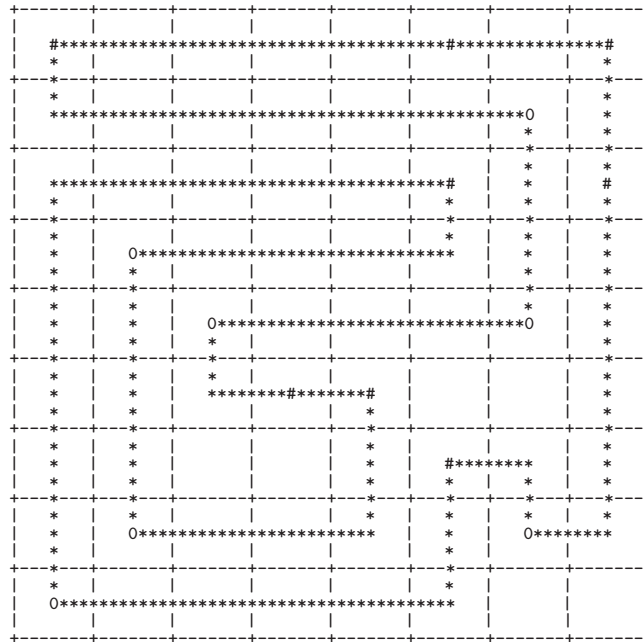


Figure 2.41: Some positions not visited

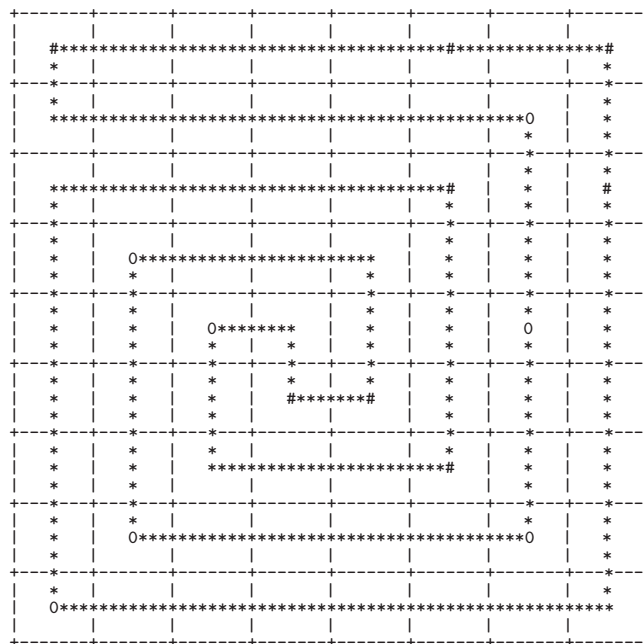


Figure 2.42: All positions visited

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



360°
thinking.

Deloitte.

Discover the truth at www.deloitte.ca/careers

© Deloitte & Touche LLP and affiliated entities.



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

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
    
```

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

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

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

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

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

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

Figure 2.44: Solving the Loop Puzzle – Variant Two

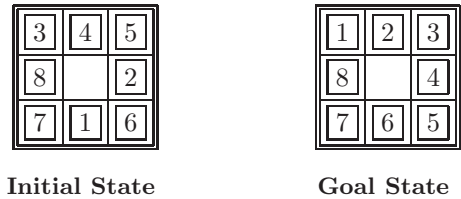


Figure 2.45: An Eight Puzzle


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.

SIMPLY CLEVER

ŠKODA


We will turn your CV into
an opportunity of a lifetime



Do you like cars? Would you like to be a part of a successful brand?
We will appreciate and reward both your enthusiasm and talent.
Send us your CV. You will be surprised where it can take you.

Send us your CV on
www.employerforlife.com

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_d1/bf/bf_d1/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.

I joined MITAS because
I wanted **real responsibility**

The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com



Month 16

I was a construction supervisor in the North Sea advising and helping foremen solve problems

Real work
International opportunities
Three work placements



