Depth-first search solves Peg Solitaire

Armando B. Matos

Technical Report Series: DCC-98-10

Departamento de Ciência de Computadores – Faculdade de Ciências
&
Laboratório de Inteligência Artificial e Cincia de Computadores

Universidade do Porto
Rua do Campo Alegre, 823 4150 Porto, Portugal
Tel: +351+2+6001672 – Fax: +351+2+6003654
http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

Depth-first search solves Peg Solitaire

Armando B. Matos

DCC & LIACC, Universidade do Porto
Rua do Campo Alegre 823, 4150 Porto, Portugal

Abstract

We invite the reader to test his skillness in previewing the behaviour of a simple, depth-first search algorithm that solves (or tries to solve) the Peg-solitaire game; the test is based on a few questions about the search tree involving aspects such as the number of nodes explored by the algorithm, the influence of the ordering of the peg moves on the efficiency of the computation and the "average branching factor" of internal nodes. We refrain from presenting at this moment the conclusions of this paper, so that the reader can fully enjoy the experiment that will be described in this report.

1 Introduction

The efficiency of depth-first search algorithms and the use of heuristics to speed up the search have always been an important practical issue in Artificial Intelligence. This paper concerns the use of a simple depth-first algorithm for the old game of Peg Solitaire and some of its variants. This is a very well known game; it is briefly explained in Appendix 1. Reference [BCG82] gives very interesting "structured" ways for solving the game. But here we are not interested in intelligent ways to solve the game but rather in the behaviour of a "stupid" program, i.e. one that does not use any heuristic to guide the search.

Appendix 2 contains a listing of a simple program in C language that tries to solve the game for several initial configurations of the board. The program uses a simple depth-first search *without any heuristics*. The pegs are considered in a top to bottom, left to right order and all possible moves are tried.

Does the program solve the game in useful time? If yes, how many board configurations are visited? If not, what strategy should we use? What is the average "branching factor" (as defined below)?

We invite the reader to test his skillness in previewing the behaviour of this simple, depth-first search by answering a few questions similar to these. It is very important to try to answer all the questions **before looking at the answers**. Do not turn to the answer of a question until you have made a genuine effort to solve it! The interested reader may find in Appendices 3 and 4 other programs so that he can to check our results or, by suitable changes, to continue the experimental research on this area.

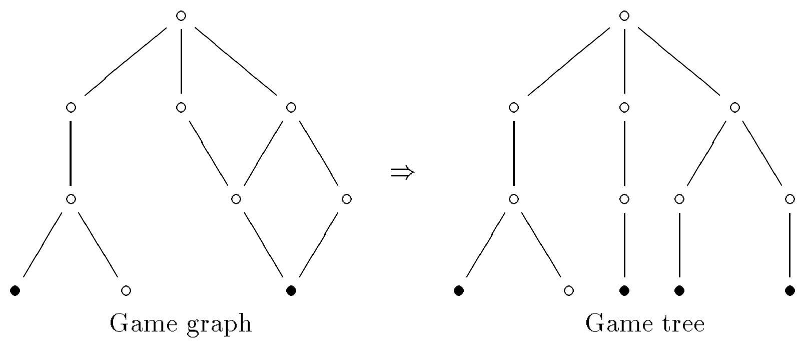


Figure 1: The graph of the game (at left) is seen by the search algorithm as a tree (at right).

Before presenting the questions in next Section, let us characterize more precisely the search tree associated with an algorithm. The search space of the game may be represented by a directed graph. The graph is acyclic because every move reduces the number of pegs by one. It is not however a tree as the reader can easily check.

Our program, being as simple as possible, does not detect repetitions of board configurations so that the *graph* should be expanded to a tree as illustrated in Figure 1.

In our case, the game graph has no cycles and the maximum depth of a node equals the number of pegs in the initial configuration minus 1.

We now define several parameters related to a search. The definitions are illustrated with the game tree represented at the right side of Figure 1 where terminal nodes marked "•" correspond to solutions.

- d: Depth of the search (or height of the tree). In this game it is one less the number of pegs. In the example $d = 3$; in the solitaire game, $d = 33 - 1 = 32$.
- s: Number of nodes visited until the first solution is found. In the example, assuming a left to right, depth-first search, we have $s = 4$ corresponding to the path from the tree to the leftmost solution ("•").
- $m_s = s - 1$: Number of moves made until the first solution is found. A move corresponds to a branch of the tree. In the example, $m_s = 3$.
- t: Total number of nodes visited. In the example, $t = 13$.
- $m_t = t - 1$: Total number of moves which equals the number of branches in the game tree. In the example, $m_t = 12$.
- i, e: Number of internal and external nodes, respectively. In the example, $i = 8$, $e = 5$.
- b: Average branching factor (number of sons) or simply *branching* of internal nodes. In the example, for the complete search, $b = (3+1+2+1+1+2+1+1)/8 = 12/8 = 1.5$. Average branching can be defined for every partial search, for instance, until finding the first solution, as

$$b = \frac{\text{Total number of moves taken so far}}{\text{Number of nonterminal nodes visited so far}}$$

– s: Number of solutions, corresponding in the game graph to the number of paths from the root to a "•" node. In the example, $s = 4$.

The number of "holes" is 33 and the number of possible boards (if there are 33 pegs available) equals $= t = 2^{33} = 8\,589\,934\,592$.

An upper bound on the number of possible moves from a board with n pegs is

$$m \le m_u = 4^{n-1} \times n!$$

Proof: each peg has at most 4 legal moves.

The time of a computation is essentially dependent on the number of moves done. Current PC's (using the GNU C compiler under the Linux operating system) typically examine between 10^5 and 10^6 moves per second.

The rest of this paper is organized as follows. Chapters 2 and 3 contain respectively 6 questions about a simple depth-first search for the solution of Peg Solitaire and the corresponding answers (your score is based on the number of correct answers. Finally some conclusions are presented in Chapter 4.

2 The questions

Computation time

Question 1 (Computation time) *The program in Appendix 2*

1. *Takes less than half second to compute a solution, exploring about 20 000 nodes (the number of nodes explored is printed by the program).*
2. *Takes about ten minutes to compute a solution, exploring about 700 million nodes.*
3. *Does not terminate within 10 hours time, after exploring more than 2×10^9 nodes.*

Complete searches

Question 2 (A complete search) *The number m_t of moves made during a complete search of the solitaire game and the corresponding number s of solutions satisfy (select the strongest statement)*

1. $m_t \ge 10^4$, $s \ge 10^2$
2. $m_t \ge 10^5$, $s \ge 10^3$
3. $m_t \ge 10^7$, $s \ge 10^4$
4. $m_t \ge 10^9$, $s \ge 10^5$

Ordering of the directions in the search

The program scans the board in a top to bottom, left to right direction. For each peg it finds, the possible moves are tried in the following order: North (N), East (E), South (S) and West (W). A move is possible (and executed) if the two next places in the corresponding direction are respectively occupied and vacant. There is no special reason for using this particular order; we have tested the program behaviour with all $24 = 4!$ possible sequences of directions. Recall that the original configuration of the solitaire game is symmetrical.

Question 3 (Influence of order) *For the 24 possible orderings of the 4 directions considered for the move of a peg (see the programs in [ABM98]),*

1. *The number of nodes examined is always the same for each of the 24 orderings.*
2. *There are exactly 24 possible numbers of examined nodes.*
3. *There are exactly 6 possible numbers of examined nodes.*
4. *There are exactly 3 possible numbers of examined nodes.*

What are the best (corresponding to less nodes explored) direction orderings?

Average number of branches

Recall that, for each search node, the branching is defined as the total number of moves taken so far divided by the number of nonterminal nodes visited so far; terminal nodes are not considered.

Question 4 (Branching) *For both the fireplace and the solitaire games, the branching b of a first solution search satisfies*

1. $2.02 \leq b \leq 2.18$
2. $1.40 \leq b \leq 1.94$
3. $4.16 \leq b \leq 4.40$
4. $16.0 \leq b \leq 31.8$

Number of noddes at a specific level

Consider a complete search in the game “fireplace”. Obviously, there is only one node at depth 0 corresponding to the initial confiuration; at depth 10 (number of pegs minus 1) there are as many nodes as there are solutions to the problem (8).

Question 5 (Number of nodes as a function of depth) *In a complete search in the game “fireplace”, how many nodes are there at depths 0, 1, 2,…,10?*

1. *1, 5, 20, 80, 350, 1272, 2532, 4860, 5854, 846, 8.*

2. *1, 5, 814, 927, 1011, 1218, 1415, 2223, 2776, 4311, 8.*

3. *1, 5, 17, 138, 217, 516, 1337, 1629, 1890, 3412, 8.*

4. *1, 5, 31, 90, 274, 618, 1840, 2017, 3099, 1126, 8.*

Number of nodes as a function of the number of sons, at a specific level

In this question the reader should guess the total number of configurations of a certain game – "fireplace" – at a certain depth – when there are 5 pegs remaining – that have 0 sons (terminal nodes), 1 son,..., 7 sons.

Question 6 (Number of nodes/number of sons) *Consider a complete search in the game "fireplace". At depth 6, that is, in boards with 5 pegs, the number of configurations having 0, 1,..., 7 sons (there are not nodes with more than 7 nodes) are respectively*

1. *101, 811, 321, 118, 44, 31, 14, 2.*
2. *110, 990, 894, 146, 356, 0, 32, 4.*
3. *80, 112, 229, 384, 128, 61,31, 1.*
4. *15, 97, 244, 1199, 889, 244,98, 7.*

3 The answers

Computation time

Question 1 (Computation time, correct answer: 1)
The program took less than half second, after making 20 278 moves.

Surprise: To my great surprise, the program solves solitaire almost immediately. ⋄

The quite unexpected results corresponding to the number of nodes visited until the first solution is found (for Peg solitaire and the other versions of the game) are summarized in Figure 3.

Surprise: Solitaire, usually considered the most challenging problem, is not the hardest for our program. "Diamond" takes much more time to solve. ⋄

Surprise: The challenging versions of a game (names ending with "$\star$" in Figure 3) are almost as hard as the non-challenging ones. In fact, the difference in the number of nodes visited never exceeds 1! ⋄

Complete searches

Question 2 (A complete search, correct answer: 4)
There are more than 2×10^9 nodes accessible from the initial position. The number of solutions exceeds 690 000.

Problem	Pegs	Nodes visited until first solution
Cross	6	12
Cross⋆	6	12
Plus	9	75
Plus⋆	9	76
Fireplace	11	5 941
Fireplace⋆	11	5 941
Up-arrow	17	17 998 001
Up-arrow⋆	17	17 998 001
Pyramid	16	797 378
Pyramid⋆	16	797 379
Diamond	24	8 528 473
Diamond⋆	24	8 528 474
Solitaire	32	20 278
Solitaire⋆	32	20 279

Figure 2: Number of nodes explored when finding the first solution

Problem	Pegs	Nodes visited	No. of solutions
Cross	6	32	4
Plus	9	580	32
Fireplace	11	15 827	8
Pyramid	16	735 033 270	10 142 448
Solitaire	32	$> 33 \times 10^9$	$> 3\,400\,000$

Figure 3: Complete search from the initial position: number of nodes visited and total number of solutions.

Figure 3 shows the number of accessible nodes and the number of solutions for 3 of the problems.

Ordering of the directions in the search

Question 3 (Influence of order, correct answer: 4)
The correct answer is: There are exactly 3 possible numbers of examined nodes. See Figure 3

Figure 3 shows that there are 3 possible number of moves: $a = 20\,275$, $b = 20\,278$ and $c = 7\,667\,769$. We have

- a occurs for the orderings NSEW, NSWE, SNEW, SNWE, NWSE and SENW.
- b occurs for the orderings NESW, NEWS, ENWS, ENSW, NWES and ESNW.
- c occurs for the other 12 orderings.

Surprise: The initial configuration of solitaire is symmetric and all 4 directions are equivalent. Some people think that, due to the symmetry of the initial configuration, the search is not

Order	Moves	Order	Moves	Order	Moves
NESW	20 278	NWES	20 278	NSWE	20 275
ESWN	7 667 769	WESN	7 667 769	SWEN	7 667 769
SWEN	7 667 769	ESNW	20 278	WENS	7 667 769
WNES	7 667 769	SNWE	20 275	ENSW	20 278
NEWS	20 278	NWSE	20 275	NSEW	20 275
EWSN	7 667 769	WSEN	7 667 769	SEWN	7 667 769
WSNE	7 667 769	SENW	20 275	EWNS	7 667 769
SNEW	20 275	ENWS	20 278	WNSE	7 667 769

Figure 4: Influence of order in the number of moves to find the first solution.

influenced by the ordering of the directions. However that is false and the reason is that the pegs are considered in a non-symmetric sequence – top to bottom, left to right – and symmetry is quickly destroyed. ⋄

Average number of branches

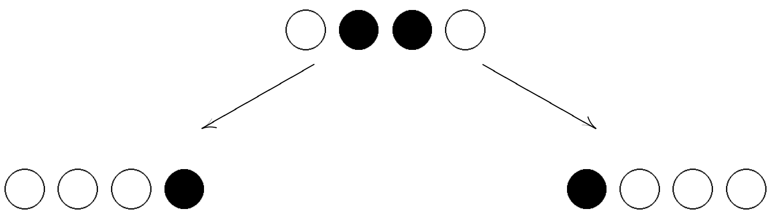
Question 4 (Branching, correct answer: 1)
For both the fireplace and the solitaire games, the branching b of a first solution search satisfies

$$2.02 \leq b \leq 2.18$$

Surprise: In fact, whenever the search involves a relatively large number of nodes, the branching factor is very nearly 2 (see Figure 3). ⋄

The following reasoning, although rather incomplete, may in part justify this claim.

Let us call a node "next to leaves" if every move from it results in a terminal configuration. Very often for those nodes there are exactly 2 possibilities for the last move (before reaching a terminal node) and for the next to the last move; the corresponding configuration and transitions are something like (for the last move and where "dead" pegs are not represented).

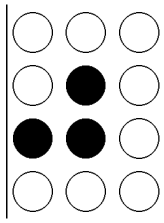


For all these nodes the branching is 2. There are obviously other, less frequent, possibilities: the node may have for instance only one son:

Problem	Nodes visited	Branching factor
Plus at first solution	75	1.9737
Plus, complete search	580	2.0939
Fireplace at first solution	5 941	2.1706
Fireplace, complete search	15 827	2.2160
Pyramid at first solution	10 142 448	2.0843
Pyramid, complete search	735 033 270	2.1876
Solitaire at first solution	20 278	2.0205

Figure 5: Branching factor for some problems.

It may also have 3 or more sons; in the following case, it has 3



But what about higher (farther from the leaves) nodes? [1] The configurations of those nodes seem to usually have at least two sons. If they have exactly 2 sons they obviously contribute to the validity of the "2 branching factor claim". If they have 3 or more sons, it is easy to see that they are much less in number than the "next to the leaves" nodes so that the branching factor is essentially determined by the "next to the leaves" nodes.

Number of noodes at a specific level

Question 5 (Number of nodes as a function of depth, correct answer: 2)
The number of configurations (or nodes) as a function of depth is

Depth	0	1	2	3	4	5	6	7	8	9	10
Nodes	1	5	20	80	350	1272	2532	4860	5854	846	8
Growth		5.00	4.00	4.00	4.37	3.63	1.99	1.92	1.20	0.14	0.0095

In the previous table we have also included the "growth" factor defined as

$$g(d) = \frac{\text{N. of nodes at depth } d}{\text{N. of nodes at depth } d-1}$$

The function $g(d)$ is relatively smooth, having a value between 4 and 5 up to depth 4 and then quickly decreasing until $d = 10$.

[1] Notice that we not talking about nodes depth (distance from the root) but about their minimum distance to a leave.

	Number of sons								
Pegs	0	1	2	3	4	5	6	7	8
11						1			
10				2	1	2			
9			2	8	6		2		2
8			20	4	26	6	10	8	6
7		10	56	112	68	88	10	6	
6	126	540	122	300	114	44	18		8
5	110	990	894	146	356		32	4	
4	2204	424	1734	30	468				
3	5396	70	388						
2	842		4						
1	8								

Figure 6: Fireplace problem: number of nodes having a given number of sons at each depth (depth=11-pegs). For clarity, when the number of nodes is 0, it is not represented. For pegs=1 there are 8 terminal nodes corresponding to the 8 solutions. Only the initial configuration has 11 pegs, having – as can be easily verified – 5 sons.

Number of branches at a specific level

Question 6 (Number of nodes/number of sons, correct answer: 2)
The number of configurations with 5 pegs as a function of the number of sons is

Number of sons	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Number of nodes	*110*	*990*	*894*	*146*	*356*	*0*	*32*	*4*

Surprise: The number of sons as a function of the number of nodes and depth seems to be quite "chaotic". Figure 3 shows this dependence for the fireplace problem. ◇

Measure your perfomance

Read the correct answers below and check the your answers, counting the total number of correct ones. As there are 4 possible answers for each question, a random selection corresponds to an expected number of correct answers of $6/4 = 1.5$ so that 2 or less correct answers corresponds to very poor perfomance.

2 or less correct answers: All your analysis efforts were futile. Perhaps you should abandon the area of of search program analysis. That might be a good advice anyway.

3 or 4 correct answers: Your expertise in this area is quite low; your correct answers are not a result of your wisdom, but your luck.

5 or 6 correct answers: You are an extremely lucky. You should use your luckiness in other area; you may for instance look for something like "fuzzy robots with non-monotonic second order meta-learning".

In a more serious vein, as explained in more detail in the next Chapter, we think that the questions proposed to the reader (or at least some of them) are intrinsically difficult so that your possibly bad score is quite comprehensible.

4 Conclusions and further research

Several competent people have tested their predictions by answering the questions described in this article. Their results were in general quite poor – and they often got very surprised when they saw the answers. But there are good reasons for this

- Trivial modifications of the program (like changing the ordering of the directions considered for the move of a peg) can have dramatic and unexpected effects on its running time.
- The search tree seems to be rather irregular; see for instance the table in Figure 3 where, for each depth, the number of nodes in function of the number of sons is represented.

We conjecture that the detailed behaviour of search algorithms for hard problems is very difficult to predict. One should be suspicious about quick predictions – even when made by experienced people – for they often turn out to be wrong. Apparently, only exhaustive and careful analysis can lead to reliable predictions. However, due to the seemingly "chaotic" behaviour of the search, we think that in many cases such analysis might be impossible.

We have been considering specific search trees of specific problems. In a more general setting we may consider the search trees associated with problems in certain *classes*; in particular it may be interesting to think about difficult instances of NP-complete problems. While many NP-complete problems seem to have a relatively small number of hard instances (an interesting example is the random 3-SAT problem discussed in [HHW96, CA96, GW96]), there are others such that every algorithm takes super-polynomial time almost everywhere; each problem which is not in P has such a "complexity core" (a hard sub-problem), see [Lynch75, BD87, ESY85]. In such cases the search trees are, by definition, large (except for a finite number of instances). We conjecture that, in this case, they are also complex, not in the sense of having a large Kolmogorov complexity ([LV94]) – the algorithm together with a particular instance are a short description of the tree – but in the computational sense; in other words we think that, for every algorithm that searches the solution of an NP-complete problem, no polynomial time algorithm can answer many of the questions related with the corresponding search tree; one such difficult question (and this is admitly a trivial observation) is clearly the following: does the tree have a leaf which is a solution?

Must the structure of search trees associated with any algorithm that decides an NP-problem, be in some sense (and for difficult instances) computationally "chaotic"? The formalization of this question and its proof or disproof, corresponds to a clarification of the relationship between the complexity of a problem and the complexity of the corresponding search tree [2] (or graph); this is clearly an area deserving further work.

[2] More generally, instead of talking about the complexity of the search tree – which only makes sense for search algorithms – we could talk about the complexity of the corresponding *computation*.

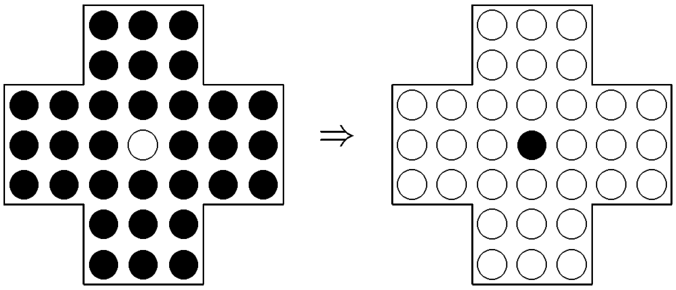


Figure 7: The Solitaire game. In the normal version, "solitaire", the final configuration must contain only one peg. In the challenging version, "solitaire$\star$", the final peg must be at the center of the board.

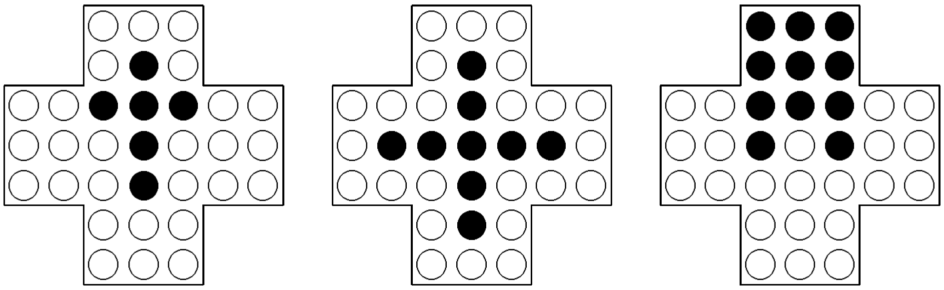


Figure 8: Left to right: initial configurations for versions "cross", "plus" and "fireplace". In the challenging versions, denoted by "cross$\star$", "plus$\star$" and "fireplace$\star$", the final peg must be at the center of the board.

Appendix 1: The rules of Peg Solitaire

Peg Solitaire is played by jumping a peg across any adjacent peg and placing it in an open space on the other side. Only horizontal and vertical moves are legal. A winning configuration is a board with only one peg. In a more challenging version of the game, the last peg must be at the center of the board.

In the "solitaire" version, the initial board has a peg in every hole except at the center; see Figure 7. Other common initial board configurations are shown in Figures 8 and 9.

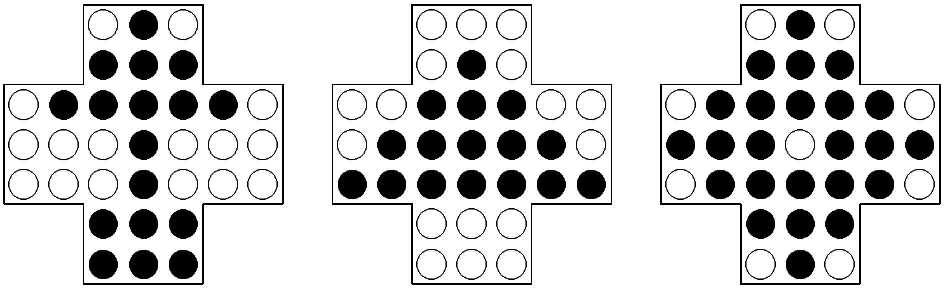


Figure 9: Left to right: initial configurations for versions "up", "pyramid" and "diamond". In the challenging versions, denoted by "up$\star$", "pyramid$\star$" and "diamond$\star$", the final peg must be at the center of the board.

Appendix 2: A Program that solves Peg Solitaire

We present a program that finds the first solution for several initial configurations of the board. Please notice that

- In the beginning of the program exactly one problem must be selected by uncommenting the corresponding "`#include`" line. The corresponding files are given at the end of the listing.
- Selecting the non-challenging or the "challenging" version is made by selecting either "`TEST1`" or "`TEST2`" in the "`move`" function.

These remarks also apply to the programs in appendices 3 and 4.

```
/* ------------------------------------
   peg game, Armando Matos 1998
   first solution with moves and boards
-------------------------------------- */

#include <stdio.h>
#define N        9

/* ---------------------------------------- */
/* - Include (only) one problem file        */
/*   by uncomenting the corresponding line   */
/* ---------------------------------------- */

/* #include "xcross" */
/* #include "xplus"  */
/* #include "xfire"  */
/* #include "xup"    */
/* #include "xpyram" */
/* #include "xdiam"  */
/* #include "xsolit" */
/* #include "xprob1" */
/* ----------------------------------- */

#define YES  1
#define NO  0

#define OCCUPIED 1
#define FREE     0
#define WALL     2

#define MAXST    35

typedef char boolean;

enum dir{NORTH,EAST,SOUTH,WEST};
int dx[]={0, 1,0,-1};
int dy[]={-1,0,1, 0};

void push(char b[N][N],int,int,enum dir);
void pop(void);
void print_solution(void);
boolean move(int);
void print_board(char b[N][N]);
```

```
long int moves=0; /* total number of moves */
int pegs=0;

main(){
  int x,y;
  int steps=0;
  for(x=0;x<N;x++)         /* count pegs */
    for(y=0;y<N;y++)
      if(b[y][x]==OCCUPIED)
        pegs++;
  printf("(%3d pegs)\n",pegs);
  print_board(b);
  system("date");
  if(move(pegs))
    print_solution();
  else
    printf("no solution\n");
  printf("\n(Total number of moves = %8d)\n",moves);
  system("date");
}

boolean move(int pegs){
  int x,y,xn,yn,xnn,ynn;
  enum dir d;

/* TEST1: solution = one peg                 */
/*
  if(pegs==1){
    return(YES);
  }
*/

/* TEST2: solution = one peg at center (4,4) */
  if(pegs==1) {
    if (b[4][4]==OCCUPIED)
      return(YES);
    else return(NO);
  }

  for(x=0;x<N;x++)
    for(y=0;y<N;y++)
      if(b[y][x]==OCCUPIED)
        for(d=NORTH;d<=WEST;d++){
          xn=x+dx[d];
          yn=y+dy[d];
          if(b[yn][xn]==OCCUPIED){
            xnn=xn+dx[d];
            ynn=yn+dy[d];
            if(b[ynn][xnn]==FREE){
              b[y][x]=FREE;          /* do move */
              b[yn][xn]=FREE;
              b[ynn][xnn]=OCCUPIED;
              pegs--;
              push(b,x,y,d);
              if(move(pegs))
                return(YES);
              b[y][x]=OCCUPIED;       /* undo move */
              b[yn][xn]=OCCUPIED;
              b[ynn][xnn]=FREE;
              pegs++;
              pop();
            }
          }
        }
  return(NO);
}
/*---------- stack -------------------------------------*/
int sp=0;
struct {
  char board[N][N];
  int vert;
  int hor;
  int dir;
} stack[MAXST];

char dirname[4][10];

void push(char b[N][N],int xm,int ym,enum dir d){
  int x,y;
  for(y=0;y<N;y++)
    for(x=0;x<N;x++)
      stack[sp].board[y][x]=b[y][x];
  stack[sp].hor=xm;
  stack[sp].vert=ym;
```

```
  stack[sp].dir=d;
  sp++;
  moves++;
  if(moves%100000==0)
    printf("(%8d moves)\n",moves);
}
void pop(void){
  sp--;
}
/*---------- print -------------------------------------*/
void print_solution(void){
  int i;
  strcpy(dirname[0],"up");
  strcpy(dirname[1],"right");
  strcpy(dirname[2],"down");
  strcpy(dirname[3],"left");
  for(i=0;i<sp;i++){
    print_board(stack[i].board);
    fflush(stdout);
    printf("[moved peg at line %2d, col %2d",
            stack[i].vert,stack[i].hor);fflush(stdout);
    printf(" in direction %s]\n",dirname[stack[i].dir]);
    fflush(stdout);
  }
}
void print_board(char b[N][N]){
  int x,y;
  printf("---------------------------------------------------\n");
  for(y=0;y<N;y++){
    for(x=0;x<N;x++){
      if(b[y][x]==1)
        printf(" #");
      else if(b[y][x]==0)
        printf(" O");
      else
        printf("  ");
    }
    printf("\n");
  }
}
```

Files for each version of the game.

```
/* FILE xsolit  */
char b[N][N]={
    {2, 2,2,2,2,2,2,2, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 1,1,1,1,1,1,1, 2},
    {2, 1,1,1,0,1,1,1, 2},
    {2, 1,1,1,1,1,1,1, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 2,2,2,2,2,2,2, 2}
  };
char *prob="solitaire";
-------------------------------------
/* FILE xcross  */
char b[N][N]={
    {2, 2,2,2,2,2,2,2, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,0,1,0,2,2, 2},
    {2, 0,0,1,1,1,0,0, 2},
    {2, 0,0,0,1,0,0,0, 2},
    {2, 0,0,0,1,0,0,0, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,2,2,2,2,2, 2}
  };
char *prob="cross";
-------------------------------------
/* FILE xplus  */
char b[N][N]={
    {2, 2,2,2,2,2,2,2, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,0,1,0,2,2, 2},
    {2, 0,0,0,1,0,0,0, 2},
```

```
    {2, 0,1,1,1,1,1,0, 2},
    {2, 0,0,0,1,0,0,0, 2},
    {2, 2,2,0,1,0,2,2, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,2,2,2,2,2, 2}
  };
char *prob="plus";
--------------------------------------
/* FILE xfire  */
char b[N][N]={
    {2, 2,2,2,2,2,2,2, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 0,0,1,1,1,0,0, 2},
    {2, 0,0,1,0,1,0,0, 2},
    {2, 0,0,0,0,0,0,0, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,2,2,2,2,2, 2}
  };
char *prob="fireplace";
--------------------------------------
/* FILE xup  */
char b[N][N]={
    {2, 2,2,2,2,2,2,2, 2},
    {2, 2,2,0,1,0,2,2, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 0,1,1,1,1,1,0, 2},
    {2, 0,0,0,1,0,0,0, 2},
    {2, 0,0,0,1,0,0,0, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 2,2,2,2,2,2,2, 2}
  };
char *prob="up-arrow";
--------------------------------------
/* FILE xpyram  */
char b[N][N]={
    {2, 2,2,2,2,2,2,2, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,0,1,0,2,2, 2},
    {2, 0,0,1,1,1,0,0, 2},
    {2, 0,1,1,1,1,1,0, 2},
    {2, 1,1,1,1,1,1,1, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,0,0,0,2,2, 2},
    {2, 2,2,2,2,2,2,2, 2}
  };
char *prob="pyramid";
--------------------------------------
/* FILE xdiam  */
char b[N][N]={
    {2, 2,2,2,2,2,2,2, 2},
    {2, 2,2,0,1,0,2,2, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 0,1,1,1,1,1,0, 2},
    {2, 1,1,1,0,1,1,1, 2},
    {2, 0,1,1,1,1,1,0, 2},
    {2, 2,2,1,1,1,2,2, 2},
    {2, 2,2,0,1,0,2,2, 2},
    {2, 2,2,2,2,2,2,2, 2}
  };
char *prob="diamond";
--------------------------------------
```

Appendix 3: A Program that finds all solutions

The following program outputs several results referred in the text, namely: moves until first solution, average branching at first solution, total number of moves, total number of solutions, non-terminal nodes, terminal nodes, branching factor and the number of nodes with a given number of sons as a function of the search depth.

```
/* ------------------------------------
```

```
   peg game, Armando Matos 1998
   no. of solutions and no. of moves
   average branching
--------------------------------------- */
#include <stdio.h>
#define N        9
#define SOLSPIC  10000
#define BASE     1000000
#define MOVESPIC 10
/* output * (x BASE)  */

/* ------------------------------------ */
/* - Include (only) one problem file    */
/* - Select one test in "move"          */
/* ------------------------------------ */

/* #include "xtest"  */
/* #include "xcross" */
/* #include "xplus"  */
/* #include "xfire"  */
/* #include "xup"    */
/* #include "xpyram" */
/* #include "xdiam"  */
/* #include "xsolit" */
/* #include "xprob1" */
/* ------------------------------------ */

#define YES  1
#define NO  0

#define N        9
#define OCCUPIED 1
#define FREE     0
#define WALL     2

#define MAXST    35
#define MAXBR    50

typedef char boolean;

enum dir{UP,RIGHT,DOWN,LEFT};
int dx[]={0, 1,0,-1};
int dy[]={-1,0,1, 0};

long int movesA=0; /* total number of moves - MSD      */
long int movesB=0; /* total number of moves - LSD      */
long int fmoves=0; /* moves until first solution       */
float fbranch;     /* branching at first solution      */
long int sols=0;   /* solutions                        */
long int nnodes=0; /* number of nonterminal nodes      */

int ipegs, pegs;
long br[MAXST][MAXBR]; /* no. branches by level        */

/* Here we don't push the board, only the move         */
void push(int,int,enum dir);
void pop(void);
void print_solution(void);
void move(int);
void all_print(void);
void first_print(void);

main(){
  int x,y;
  int steps=0;
  pegs=0;
  for(x=0;x<N;x++)         /* count pegs */
    for(y=0;y<N;y++)
      if(b[y][x]==OCCUPIED)
        pegs++;
  ipegs=pegs;
  printf("(%3d pegs)\n",ipegs);
  for(x=0;x<MAXST;x++)   /* initialize no. of branches */
    for(y=0;y<MAXBR;y++)
      br[x][y]=0;
  system("date");
  move(pegs);
  system("date");
  all_print();
}

void move(int pegs){
  int x,y,xn,yn,xnn,ynn;
  enum dir d;
  boolean count;
  long branches; /* branches at this node                      */
/*----------------------- test selection ----------------------*/
/* TEST1: solution = one peg                                 */
   if(pegs==1){
    br[ipegs-pegs][0]+=1;
```

```
    sols++;
    if(fmoves==0){
      fmoves=movesB;
      fbranch=(float)fmoves/(float)nnodes;
      first_print();
    }
    if(sols%SOLSPIC==0)
      printf("   (%8d solutions)\n",sols);
    return;
  }

/* TEST2: solution = one peg at center (4,4) */
/*
  if(pegs==1) {
    br[ipegs-pegs][0]+=1;
    if (b[4][4]==OCCUPIED){
      sols++;
      if(fmoves==0){
        fmoves=movesB;
        fbranch=(float)fmoves/(float)nnodes;
        first_print();
      }
      if(sols%SOLSPIC==0)
        printf("   (%8d solutions)\n",sols);
      return;
    }
    else
      return;
  }
*/
/*--------------- end of test selection ----------------------*/
  count=YES;
  branches=0;     /* branches for this board  */
  for(x=0;x<N;x++)
    for(y=0;y<N;y++){
      if(b[y][x]==OCCUPIED)
        for(d=UP;d<=LEFT;d++){
          xn=x+dx[d];
          yn=y+dy[d];
          if(b[yn][xn]==OCCUPIED){
            xnn=xn+dx[d];
            ynn=yn+dy[d];
            if(b[ynn][xnn]==FREE){
              branches++;
              b[y][x]=FREE;          /* do move */
              b[yn][xn]=FREE;
              b[ynn][xnn]=OCCUPIED;
              pegs--;
              movesB++;
              if(movesB==BASE){
                movesB=0;
                movesA++;
              }
              if(count)
                nnodes++;
              count=NO;
              if(movesA%MOVESPIC==0 && movesB==0)
                printf("(moves: %8d (x BASE), %1.4f branching)\n",
                        movesA,
                        (float)(BASE*movesA+movesB)/(float)nnodes);
              move(pegs);
              b[y][x]=OCCUPIED;       /* undo move */
              b[yn][xn]=OCCUPIED;
              b[ynn][xnn]=FREE;
              pegs++;
            }
          }
        }
    }
  br[ipegs-pegs][branches]+=1;
  return;
}
/*-----------------------------------------------------------------*/
void line(void);
void branching_print(void);

void all_print(void){
  printf("\n");
  line();
  printf("\nCOMPLETE SEARCH");
  printf("\nTotal number of moves              = %d,%d",
                                              movesA,movesB);
  printf("\nTotal number of solutions          = %8d",sols);
  printf("\nNon terminal nodes                 = %8d",nnodes);
  printf("\nTerminal nodes                     = %8d",(movesB+1)-nnodes);
```

```
  printf("\nAverage branching                    = %8.4f\n",
                                (float)movesB/(float)nnodes);
  line();
  printf("\n");
  branching_print();
}
void first_print(void){
  printf("\n");
  line();
  printf("\nSEARCH UNTIL FIRST SOLUTION");
  printf("\nMoves until first solution           = %8d",fmoves);
  printf("\nAverage branching at first solution = %8.4f\n",
                                fbranch);
  line();
  printf("\n");
}
void branching_print(void){
  int dep, bra, maxbr=0;
  /* find maximum branching */
  for(dep=0;dep<MAXST;dep++)
    for(bra=0;bra<MAXBR;bra++)
      if(br[dep][bra]!=0 && bra>maxbr)
        maxbr=bra;
  line();
  printf("\n pegs ");
  for(bra=0;bra<=maxbr;bra++)
    printf("%6d",bra);
  printf("\n       ");
  for(bra=0;bra<=maxbr;bra++)
    printf("------");
  for(dep=0;dep<ipegs;dep++){
    printf("\n");
    printf("%2d     ",ipegs-dep);
    for(bra=0;bra<=maxbr;bra++){
      long t=br[dep][bra];
      if(t!=0)
        printf("%6d",t);
      else
        printf("      ");
    }
  }
  printf("\n");
  line();
  printf("\n");
}
void line(void){
  printf("----------------------------------------------");
}
```

Appendix 4: A program to study the dependence on the ordering of the directions

This program has an extra parameter that specifies the ordering of the firections in the search. The corresponding command has the form

```
peg <order>
```

where "`peg`" is the name of the program and "`<order>`" is a 4 character word that specifies a permutation of the 4 directions. Each character may be either "`n`" (north), "`e`" (east), "`s`" (south) or "`w`" (west). For instance, with the call

```
peg nswe
```

possible moves are tried (in a depth-first search) in the order: north (that is, the peg, if possible, jumps in the north or up direction), south, west and east.

```
/* ------------------------------------
   peg game, Armando Matos 1998
```

```
   influence of direction ordering
--------------------------------------- */
#include <stdio.h>
#define N         9
#define SOLSPIC   10000
#define BASE      1000000
#define MOVESPIC 10
/* output * (x BASE)  */

/* ----------------------------------- */
/* - Include (only) one problem file    */
/* - Select one test in "move"          */
/* ----------------------------------- */

/* #include "xtest"  */
/* #include "xcross" */
/* #include "xplus"  */
/* #include "xfire"  */
/* #include "xup"    */
/* #include "xpyram" */
/* #include "xdiam"  */
/* #include "xsolit" */
/* #include "xprob1" */
/* ----------------------------------- */

#define YES  1
#define NO  0

#define N        9
#define OCCUPIED 1
#define FREE     0
#define WALL     2

#define MAXST    35
#define MAXBR    50

typedef char boolean;

enum dir{UP,RIGHT,DOWN,LEFT};
int dx[4];
int dy[4];

long int movesA=0; /* total number of moves - MSD      */
long int movesB=0; /* total number of moves - LSD      */
long int fmovesA=0,
         fmovesB=0;/* moves until first solution       */
float fbranch;     /* branching at first solution      */
long int sols=0;   /* solutions                        */
long int nnodes=0; /* number of nonterminal nodes      */

int ipegs, pegs;

/* Here we don't push the board, only the move         */
void push(int,int,enum dir);
void pop(void);
void print_solution(void);
void move(int);
void all_print(void);
void first_print(void);
void  set_directions(int n, char *p[]);

main(int argc, char* argv[]){
  int x,y;
  int steps=0;
  set_directions(argc, argv);
  pegs=0;
  for(x=0;x<N;x++) /* count pegs */
    for(y=0;y<N;y++)
      if(b[y][x]==OCCUPIED) pegs++;
  ipegs=pegs;
  printf("%3d pegs, direction %s\n",ipegs,argv[1]);
  system("date");
  move(pegs);
  system("date");
  all_print(); }

void move(int pegs){
  int x,y,xn,yn,xnn,ynn;
  enum dir d;
  boolean count;

/*----------------------- test selection ----------------------*/
/* TEST1: solution = one peg                                 */
   if(pegs==1){
    sols++;
    if(fmovesA==0 && fmovesB==0){
      fmovesA=movesA;
      fmovesB=movesB;
      fbranch=(float)fmovesA/(float)nnodes;
      first_print();
    }
```

```
    if(sols%SOLSPIC==0)
      printf("   (%8d solutions)\n",sols);
    return;
  }

/* TEST2: solution = one peg at center (4,4) */
/*
  if(pegs==1) {
    if (b[4][4]==OCCUPIED){
      sols++;
    if(fmovesA==0 && fmovesB==0){
        fmovesA=movesA;
        fmovesB=movesB;
        fbranch=(float)fmovesA/(float)nnodes;
        first_print();
      }
      if(sols%SOLSPIC==0)
        printf("   (%8d solutions)\n",sols);
      return;
    }
    else
      return;
  }
*/
/*--------------- end of test selection ----------------------*/
  count=YES;
  for(x=0;x<N;x++)
    for(y=0;y<N;y++){
      if(b[y][x]==OCCUPIED)
        for(d=UP;d<=LEFT;d++){
          xn=x+dx[d];
          yn=y+dy[d];
          if(b[yn][xn]==OCCUPIED){
            xnn=xn+dx[d];
            ynn=yn+dy[d];
            if(b[ynn][xnn]==FREE){
              b[y][x]=FREE;          /* do move */
              b[yn][xn]=FREE;
              b[ynn][xnn]=OCCUPIED;
              pegs--;
              movesB++;
              if(movesB==BASE){
                movesB=0;
                movesA++;
              }
              if(count)
                nnodes++;
              count=NO;
              if(movesA%MOVESPIC==0 && movesB==0){
                printf("(moves: %8d (x BASE), %1.4f branching) -- ",
                        movesA,
                        (float)(BASE*movesA+movesB)/(float)nnodes);
                fflush(stdout);
                system("date");
              }
              move(pegs);
              b[y][x]=OCCUPIED;       /* undo move */
              b[yn][xn]=OCCUPIED;
              b[ynn][xnn]=FREE;
              pegs++;
            }
          }
        }
    }
  return;
}
/*------------------------------------------------------------------*/
void error_message(void);
int xval(char), yval(char);
int direction(char);
int deltax[] = { 0,1,0,-1};
int deltay[] = {-1,0,1, 0};
void  set_directions(int n, char *p[]){
  int i;
  if(n!=2)
    error_message();
  if(strlen(p[1])!=4)
    error_message();
  for(i=0;i<4;i++){
    dx[i]=xval(p[1][i]);
    dy[i]=yval(p[1][i]);
  }
}
```

```
int xval(char c){
  int d=direction(c);
  if(d<0)
    error_message();
  return(deltax[d]);
}
int yval(char c){
  int d=direction(c);
  if(d<0)
    error_message();
  return(deltay[d]);
}
int direction(char c){
  if(c=='n')
    return(0);
  if(c=='e')
    return(1);
  if(c=='s')
    return(2);
  if(c=='w')
    return(3);
  return(-1);
}
void error_message(void){
  printf("usage: pegd dddd where d in {n,e,s,w}\n");
  exit(1);
}
/*----------------------------------------------------------------*/
void line(void);
void all_print(void){
  printf("\n");
  line();
  printf("\nCOMPLETE SEARCH");
  printf("\nTotal number of moves                = %dxBASE+%d",
                                                movesA,movesB);
  printf("\nTotal number of solutions            = %8d",sols);
  printf("\nNon terminal nodes                   = %8d",nnodes);
  printf("\nTerminal nodes                       = %8d",(movesB+1)-nnodes);
  printf("\nAverage branching                    = %8.4f\n",
                                 (float)movesB/(float)nnodes);
  line();
  printf("\n");
}
void first_print(void){
  printf("\n");
  line();
  printf("\nSEARCH UNTIL FIRST SOLUTION");
  printf("\nMoves until first solution           =
             %8dM+%8d",fmovesB,fmovesA);
  printf("\nAverage branching at first solution = %8.4f\n",
                                 fbranch);
  line();
  printf("\n");
}
void line(void){
  printf("----------------------------------------------");
}
```

References

[BCG82] E. Berlekamp, J. Conway and R. Guy, *Winning Ways for your Mathematical Plays*, Volume 2: Games in Particular, Academic Press, 1982.

[BD87] R. B. Book and D. Du, *The existence and density of generalized complexity cores*, J. ACM, **34** 3 (July 1987), pp 718-730.

[CA96] J. M. Crawford and L. D. Auton, *Experimental results on the crossover point in random 3-SAT*, Artificial Intelligence, **81** 1-2 (1996), pp 31-57.

[ESY85] S. Even, A. L. Selman and Y. Yacobi, *Hard-core theorems for complexity classes*, J. ACM, **32**, 1 (January 1985), pp 205-217.

[GW96] I. P. Gent and T. Walsh, *The satisfiability constraint gap*, Artificial Intelligence, **81** 1-2 (1996), pp 59-80.

[HHW96] T. Hogg, B. A. Huberman and C. P. Williams, *Phase transitions and the search problem (Editorial)*, Artificial Intelligence, **81** 1-2 (1996), pp 1-15.

[LV94] M. Li and P. M. B. Vitàny, *An Introduction to Kolmogorov Complexity and its Application*, Springer-Verlag, Texts and Monographs in Computer Science, 1994,.

[Lynch75] N. Lynch, *On reducibility to complex or sparse sets*, J. ACM, **3**, 22 (July 1975), pp 341-345.

[ABM98] A. B. Matos *Depth-first search solves Peg-solitaire*, Technical Report, Department of Computer Science, Faculty of Sciences, University of Oporto, 1998.

[Ravi97] B. Ravikumar, *Peg-Solitaire, String Rewriting Systems and Finite Automata*, ISAAC: 8th International Symposium on Algorithms and Computation, 1997.