Introduction

The purpose of this project is to experiment with depth-first search to find solutions to puzzles. The type of puzzle considered is Peg Solitaire, usually played on a flat board with an initial configuration, rules for moving and removing pegs, and an objective for terminating the game.

The approach taken here is to explore a Maple software architecture that can be used in a flexible manner to explore this type of game. Maple software used for exploration is included in the Appendices and is available in the author's eden em13 directory.

Software architecture issues

Maple is a procedural language. The software architecture approach that is often used to solve a problem is to decide on a structure to represent the data of the problem, then to use a series of procedures to act on that data structure leading to a solution to the problem. This is not the only software architecture possible. Object-oriented programming is an approach where objects are defined that encapsulate both data and procedure. Communication with the objects is via methods defined for the object class.

Maple does not support object-oriented programming in the traditional way so an approach is taken here that seeks to separate data structure from procedural structure.

Software architecture for depth-first search

Depth-first search is a family of algorithms that seeks to solve a problem using forward search and backtracking. The algorithm can be specified without reference to the specific problem being solved. Indeed, the main procedure used for the current investigation is shown in Table 1. Note that the implementation completely hides all reference to data structure. The algorithm is defined using a set of procedures that are passed as arguments to the main control program. For each problem considered, the experimenter implements the given procedures and invokes the main control program to solve the problem.

A note on testing

Appendix 1 contains the Maple source code for DFScontrol as well as a set of procedures for testing the algorithm interactively. These procedures allow the user to interactively step through the algorithm supplying whatever data is desired.

Table 1

```
DFScontrol := proc(Init,IsTerm,NextProg,NextAlt,NextBack)
        local stepSuccess,backSuccess,termSuccess;
        Init();
        if IsTerm() then return true; end if;
        stepSuccess := true; termSuccess := false;
        while stepSuccess and not termSuccess do
                stepSuccess := NextProg();
                if not stepSuccess then
                        stepSuccess := NextAlt();
                end if;
                if not stepSuccess then
                        # Need to backtrack until success
                        backSuccess := true;
                        while backSuccess and not stepSuccess do
                                backSuccess := NextBack();
                                if backSuccess then
                                        stepSuccess := NextAlt();
                                end if;
                        end do;
                end if;
                if stepSuccess then
                        termSuccess := IsTerm();
                end if;
        end do;
        return termSuccess;
end proc:
```

Sample application: Depth-first search in a directed graph

A typical application for depth-first search is for finding paths in a directed graph. Software for this application is included in Appendix 2. This application was used to verify the control architecture. A problem for path algorithms in graphs is to ensure that the path does not loop back through nodes previously on the path. The implementation takes care of this problem but hides this problem from the depth-first search control procedure because this issue is not of concern in all depth-first search procedures.

The Peg Solitaire application: Experimental Results

The Peg Solitaire Maple code is included in Appendix 3. Some problems were not solved due to limitations on the performance of the algorithm. A Timer was added to the code to request a halt to the algorithm if no solution was found in a specified number of procedure invocations.

Rectangular board experiments

Various experiments were performed with rectangular board with initial configuration of one "hole" and the objective to find a final configuration with only one peg remaining. Here are some results obtained:

Configuration	**Experimental Results**
Board: 4x4, hole at [1,2]	One peg left
Board: 4x4, hole in other locations	Two pegs left
Board: 4x5, hole at [1,2]	One peg left
Board: 4x5, hole at [2,3]	One peg left
Board: 4x5, hole in other locations	Two pegs left
Board: 5x5, hole at [3,3]	Three pegs left

Standard board experiments

Two standard board configurations were considered:

American						
		1	1	1		
		1	1	1		
1	1	1	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1
		1	1	1		
		1	1	1		

European						
		1	1	1		
	1	1	1	1	1	
1	1	1	1	1	1	1
1	1	1	0	1	1	1
1	1	1	1	1	1	1
	1	1	1	1	1	
		1	1	1		

The results of this experiment were:

Configuration	**Experimental Results**
American board	One peg left in center hole after 50525 function calls
European board	Two pegs unsuccessful after 500000 function calls, Three pegs successful after 30324 function calls

Toroidal board experiments

Here the board was considered to be on the surface of a torus so that pegs could jump "around the edges" of the board. This means that all locations on the boards are equivalent in the sense that a starting "hole" can be placed in any position and an equivalent game will result.

Here are some results for toroidal boards:

Configuration	**Experimental Results**
Board: 3x3	One peg left in hole position
Board: 4x4	One peg left in hole position
Board: 5x5	One peg left in hole position

It may be interesting to note that the 5x5 board solution required 3120 function calls and arrived at the following sequence:

Cell Position				
1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Start				
0	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1
1	1	1	1	1

Finish				
1	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
0	0	0	0	0

Plays are $a(1) - b(1) - c(0) \rightarrow a(0) - b(0) - c(1)$

No.	**a**	**b**	**c**
1	3	2	1
2	5	1	2
3	11	6	1
4	1	2	3
5	3	4	5
6	8	7	6

No.	**a**	**b**	**c**
7	10	6	7
8	12	7	2
9	14	15	11
10	16	21	1
11	1	2	3
12	17	22	2

No.	**a**	**b**	**c**
13	2	3	4
14	4	5	1
15	18	23	3
16	19	24	4
17	4	3	2
18	2	1	5

No.	**a**	**b**	**c**
19	25	20	15
20	15	11	12
21	12	13	14
22	14	9	4
23	4	5	1

Appendix 1: DFS.txt

```
# Experimental Math Rocks
# Math-640 Project: Explore Solitaire games using Depth First Search (DFS)
# Phil Benjamin

# Control flow program and test program

HelpDFS := proc()
        print(`DFScontrol()`);
end proc:

# General DFS Control

# All arguments to this proc are procs applied to the problem
# All procs use global variables (thus separating data from control)

DFScontrol := proc(Init,IsTerm,NextProg,NextAlt,NextBack)
        local stepSuccess,backSuccess,termSuccess;
        Init();
        if IsTerm() then return true; end if;
        stepSuccess := true; termSuccess := false;
        while stepSuccess and not termSuccess do
                stepSuccess := NextProg();
                if not stepSuccess then
                        stepSuccess := NextAlt();
                end if;
                if not stepSuccess then
                        # Need to backtrack until success
                        backSuccess := true;
                        while backSuccess and not stepSuccess do
                                backSuccess := NextBack();
                                if backSuccess then
                                        stepSuccess := NextAlt();
                                end if;
                        end do;
                end if;
                if stepSuccess then
                        termSuccess := IsTerm();
                end if;
```

```
        end do;
        return termSuccess;
end proc:

# Test procs
with(Maplets[Examples]):
TestGetInput := proc(myMsg,myTitle)
        return parse(GetInput(myMsg,title=myTitle));
end proc:
TestMsg := proc(myMsg)
        return Message(myMsg,title="Note",type=information);
end proc:

# Test DFScontrol
TestGlobalState := "":
TestDFScontrol := proc()
        return DFScontrol(TestInit,TestIsTerm,
        TestNextProg,TestNextAlt,TestNextBack);
end proc:

TestInit := proc()
        global TestGlobalState;
        TestGlobalState := GetInput("Init:");
end proc:
TestIsTerm := proc()
        global TestGlobalState;
        return parse(GetInput(cat("IsTerm: ",TestGlobalState)));
end proc:
TestNextProg := proc()
        global TestGlobalState;
        local NewGlobalState;
        NewGlobalState := GetInput("NextProg:",value=TestGlobalState);
        if NewGlobalState <> TestGlobalState then
                TestGlobalState := NewGlobalState;
                return true;
        end if;
        return false;
end proc:

TestNextAlt := proc()
```

```
        global TestGlobalState;
        local NewGlobalState;
        NewGlobalState := GetInput("NextAlt:",value=TestGlobalState);
        if NewGlobalState <> TestGlobalState then
                TestGlobalState := NewGlobalState;
                return true;
        end if;
        return false;
end proc:

TestNextBack := proc()
        global TestGlobalState;
        local NewGlobalState;
        NewGlobalState := GetInput("NextBack:",value=TestGlobalState);
        if NewGlobalState <> TestGlobalState then
                TestGlobalState := NewGlobalState;
                return true;
        end if;
        return false;
end proc:
```

Appendix 2: DFSgraphPath.txt

```
# Experimental Math Rocks
# Math-640 Project: Explore Solitaire games using Depth First Search (DFS)
# Phil Benjamin

# DFS Application: Find Path in Graph from Source to Sink
# Prefix for this application: GR

# Note: directed graph defined by edge lists, not edge sets!

# Global variables
GR := [[],[8,7],[8,7,2,5],[],[8,2,7,1],[10,9,2,3],[8,2,3],[7,6,3]]:
GRstart := 8:
GRend := 1:
GRpath := []:

# Packages
with(ListTools):

# Control program
GRpathControl := proc()
        local success;
        success := DFScontrol(GRinit,GRisTerm,
        GRnextProg,GRnextAlt,GRnextBack);
        if success then print(GRpath);
        else print(`No path`);
        end if;
end proc:

GRinit := proc()
        global GRpath,GRstart;
        GRpath := [GRstart];
end proc:

GRisTerm := proc()
        global GRpath,GRend;
        return GRpath[nops(GRpath)] = GRend;
end proc:

GRnextProg := proc()
```

```
        global GR,GRpath;
        local n; # length of path
        local i; # current end node
        local j; # next node (maybe)
        local k; # index of next node
        n := nops(GRpath);
        if n = 0 then return false; end if;
        i := GRpath[n];
        for k from 1 to nops(GR[i]) do
                j := GR[i][k];
                if Search(j,GRpath) = 0 then
                        GRpath := [op(GRpath),j];
                        return true;
                end if;
        end do;
        return false;
end proc:

GRnextAlt := proc()
        global GR,GRpath;
        local n; # length of path
        local iPrev; # next-to-last in path
        local iCurr; # last in path
        local j; # next node (maybe)
        local k; # index of next node
        n := nops(GRpath);
        if n < 2 then return false; end if;
        iPrev := GRpath[n-1]; iCurr := GRpath[n];
        k := Search(iCurr,GR[iPrev]);
        if k = 0 or k = nops(GR[iPrev]) then
                return false;
        end if;
        for k from k+1 to nops(GR[iPrev]) do
                j := GR[iPrev][k];
                if Search(j,GRpath) = 0 then
                        GRpath := [op(1..n-1,GRpath),j];
                        return true;
                end if;
        end do;
        return false;
```

```
end proc:

GRnextBack := proc()
        global GRpath;
        local n;
        n := nops(GRpath);
        if n < 2 then return false; end if;
        GRpath := [op(1..n-1,GRpath)];
        return true;
end proc:
```

Appendix 3: DFSpegSolitaire.txt

```
# Experimental Math Rocks
# Math-640 Project: Explore Solitaire games using Depth First Search (DFS)
# Phil Benjamin

# DFS Application: Find Solution to Peg Solitaire Game
# Prefix for this application: PS

# Note: board is defined as list of
#       -1 out of bounds location
#       0 empty location
#       1 occupied location
# Moves of form: [1,1,0] --> [0,0,1] where locations are
#       in a row
#       in a column

# Global game parameters
PSboard := []:
PSnrow := 0:
PSncol := 0:
PSnwin := 0:

# Global play parameters
PSplay := []: # Members are triples [i,j,k] where [1,1,0] --> [0,0,1]
# Note: possible plays are searched in this order:
#       by row, left-to-right
#       by col, top-to-bottom
#       by row, right-to-left
#       by col, bottom-to-top
PSpotent := []: # List of potential moves
# Each element of PSpotent is a list of possible moves
#       at this step of the game
PStimer := 0: # number of moves left, end at 0
PSdbg := 0: # turn on debugging for initial moves

# Build board and initialize play parameters
PSinit := proc()
        global PSboard,PSnrow,PSncol,PSnwin;
        global PSplay,PSpotent;
        global PStimer,PSdbg;
```

```
        local n,i,j;

        # Build rectangular board
        PSnrow := 5;
        PSncol := 5;
        n := PSnrow * PSncol;
        PSboard := [1$n];

        # Place forbidden squares (if any)

        # Place empty squares
        PSboard[PSrc2i(1,1)] := 0;

        # Initialize play parameters
        PSnwin := 1;
        PSplay := [];
        PSpotent := [];
        PStimer := 10000;
        PSdbg := 3;
end proc:

# Packages
with(ListTools):

# Debugging tool
PSDBG := proc(PSproc)
        global PSdbg;
        local n;
        if PSdbg = false then return; end if;
        n := nops(PSplay);
        if PSdbg < n then return; end if;
        print(cat(PSproc," n: ",convert(n,string)));
        if n > 0 then
                print(cat(PSproc," PSpotent: ",convert(PSpotent[n],string)));
                print(cat(PSproc," PSplay: ",convert(PSplay[n],string)));
        end if;
end proc:

# Standard control program
PScontrol := proc()
```

```
        local success;
        success := DFScontrol(PSinit,PSisTerm,
        PSprog,PSalt,PSback);
        if success then print(`Success`);
        else print(`Fail`);
        end if;
end proc:

# Toroidal control program
PSTcontrol := proc()
        local success;
        success := DFScontrol(PSinit,PSisTerm,
        PSTprog,PSalt,PSback);
        if success then print(`Success`);
        else print(`Fail`);
        end if;
end proc:

PSisTerm := proc()
        global PSboard,PSnwin;
        local b1;
        b1 := SearchAll(1,PSboard);
        if nops([b1]) > PSnwin then return false; end if;

        # Could test for perfect win here ...
        return PSboard[PSrc2i(1,1)] = 1;
#       return true;
end proc:

# Utility function: r,c --> i (row and column to board index)
# Return 0 if out of bounds
PSrc2i := proc(r,c)
        global PSnrow, PSncol;
        if r < 1 or r > PSnrow then return 0; end if;
        if c < 1 or c > PSncol then return 0; end if;
        return PSncol*(r-1) + c;
end proc:

# Toriodal version of PSrc2i: never returns 0!
PSTrc2i := proc(r,c)
```

```
        global PSnrow, PSncol;
        local r1,c1;
        r1 := 1 + ((r - 1) mod PSnrow);
        c1 := 1 + ((c - 1) mod PSncol);
        return PSncol*(r1-1) + c1;
end proc:

# Standard version
PSprog := proc()
        global PSboard,PSnrow,PSncol,PSplay,PSpotent,PStimer;
        local PSpotentNew,play;
        local r1,c1,r2,c2,r3,c3,i1,i2,i3; # indices for potential moves
        local n;

        # Check timer first
        if PStimer = 0 then return false; end if;
        PStimer := PStimer - 1;

        # Build list of potential moves
        PSpotentNew := [];
        for r1 from 1 to PSnrow do
        for c1 from 1 to PSncol do
                i1 := PSrc2i(r1,c1);
                if PSboard[i1]<>1 then next; end if;

                # Try by row, left-to-right
                r2 := r1; c2 := c1 + 1; r3 := r1; c3 := c2 + 1;
                i2 := PSrc2i(r2,c2); i3 := PSrc2i(r3,c3);
                if i2>0 and i3>0 and PSboard[i2]=1 and PSboard[i3]=0 then
                        PSpotentNew := [op(PSpotentNew),[i1,i2,i3]];
                end if;

                # Try by col, top-to-bottom
                r2 := r1 + 1; c2 := c1; r3 := r2 + 1; c3 := c2;
                i2 := PSrc2i(r2,c2); i3 := PSrc2i(r3,c3);
                if i2>0 and i3>0 and PSboard[i2]=1 and PSboard[i3]=0 then
                        PSpotentNew := [op(PSpotentNew),[i1,i2,i3]];
                end if;

                # Try by row, right-to-left
```

```
                r2 := r1; c2 := c1 - 1; r3 := r1; c3 := c2 - 1;
                i2 := PSrc2i(r2,c2); i3 := PSrc2i(r3,c3);
                if i2>0 and i3>0 and PSboard[i2]=1 and PSboard[i3]=0 then
                        PSpotentNew := [op(PSpotentNew),[i1,i2,i3]];
                end if;

                # Try by col, bottom-to-top
                r2 := r1 - 1; c2 := c1; r3 := r2 - 1; c3 := c2;
                i2 := PSrc2i(r2,c2); i3 := PSrc2i(r3,c3);
                if i2>0 and i3>0 and PSboard[i2]=1 and PSboard[i3]=0 then
                        PSpotentNew := [op(PSpotentNew),[i1,i2,i3]];
                end if;

        end do; end do;

        if PSpotentNew = [] then return false; end if;
        play := PSpotentNew[1];

        # Update PSboard, PSplay, and PSpotent
        PSboard[play[1]] := 0; PSboard[play[2]] := 0; PSboard[play[3]] := 1;
        PSplay := [op(PSplay),play];
        PSpotent := [op(PSpotent),PSpotentNew];
PSDBG("PSprog");
        return true;
end proc:

# Toroidal version
PSTprog := proc()
        global PSboard,PSnrow,PSncol,PSplay,PSpotent,PStimer;
        local PSpotentNew,play;
        local r1,c1,r2,c2,r3,c3,i1,i2,i3; # indices for potential moves
        local n;

        # Check timer first
        if PStimer = 0 then return false; end if;
        PStimer := PStimer - 1;

        # Build list of potential moves
        PSpotentNew := [];
        for r1 from 1 to PSnrow do
```

```
for c1 from 1 to PSncol do
        i1 := PSTrc2i(r1,c1);
        if PSboard[i1]<>1 then next; end if;

        # Try by row, left-to-right
        r2 := r1; c2 := c1 + 1; r3 := r1; c3 := c2 + 1;
        i2 := PSTrc2i(r2,c2); i3 := PSTrc2i(r3,c3);
        if i2>0 and i3>0 and PSboard[i2]=1 and PSboard[i3]=0 then
                PSpotentNew := [op(PSpotentNew),[i1,i2,i3]];
        end if;

        # Try by col, top-to-bottom
        r2 := r1 + 1; c2 := c1; r3 := r2 + 1; c3 := c2;
        i2 := PSTrc2i(r2,c2); i3 := PSTrc2i(r3,c3);
        if i2>0 and i3>0 and PSboard[i2]=1 and PSboard[i3]=0 then
                PSpotentNew := [op(PSpotentNew),[i1,i2,i3]];
        end if;

        # Try by row, right-to-left
        r2 := r1; c2 := c1 - 1; r3 := r1; c3 := c2 - 1;
        i2 := PSTrc2i(r2,c2); i3 := PSTrc2i(r3,c3);
        if i2>0 and i3>0 and PSboard[i2]=1 and PSboard[i3]=0 then
                PSpotentNew := [op(PSpotentNew),[i1,i2,i3]];
        end if;

        # Try by col, bottom-to-top
        r2 := r1 - 1; c2 := c1; r3 := r2 - 1; c3 := c2;
        i2 := PSTrc2i(r2,c2); i3 := PSTrc2i(r3,c3);
        if i2>0 and i3>0 and PSboard[i2]=1 and PSboard[i3]=0 then
                PSpotentNew := [op(PSpotentNew),[i1,i2,i3]];
        end if;

end do; end do;

if PSpotentNew = [] then return false; end if;
play := PSpotentNew[1];

# Update PSboard, PSplay, and PSpotent
PSboard[play[1]] := 0; PSboard[play[2]] := 0; PSboard[play[3]] := 1;
PSplay := [op(PSplay),play];
```

```
        PSpotent := [op(PSpotent),PSpotentNew];
PSDBG("PSTprog");
        return true;
end proc:

PSalt := proc()
        global PSboard,PSplay,PSpotent,PStimer;
        local play,k,n;

        # Check timer first
        if PStimer = 0 then return false; end if;
        PStimer := PStimer - 1;

        n := nops(PSplay); # also nops(PSpotent)
        if n = 0 then return false; end if;
        play := PSplay[n];
        k := Search(play,PSpotent[n]);
        if k = 0 or k = nops(PSpotent[n]) then return false; end if;

        # Undo current play
        PSboard[play[1]] := 1; PSboard[play[2]] := 1; PSboard[play[3]] := 0;

        # Update PSboard and PSplay with new play
        play := PSpotent[n][k+1];
        PSboard[play[1]] := 0; PSboard[play[2]] := 0; PSboard[play[3]] := 1;
        PSplay := [op(1..n-1,PSplay),play];
PSDBG("PSalt");
        return true;
end proc:

PSback := proc()
        global PSboard,PSplay,PSpotent,PStimer;
        local play,k,n;

        # Check timer first
        if PStimer = 0 then return false; end if;
        PStimer := PStimer - 1;

        n := nops(PSplay); # also nops(PSpotent)
        if n = 0 then return false; end if;
```

```
        play := PSplay[n];

        # Undo current play, PSplay, PSpotent
        PSboard[play[1]] := 1; PSboard[play[2]] := 1; PSboard[play[3]] := 0;
        PSplay := [op(1..n-1,PSplay)];
        PSpotent := [op(1..n-1,PSpotent)];
        n := nops(PSplay); # also nops(PSpotent)
PSDBG("PSback");
        return true;
end proc:

# Experimental Results
# Note: Experiments limited to 100000 function calls to prog,alt,back
# Board: 4x4 one hole:
#       1 left if hole at [1,2]
#       2 left if hole at other places
# Board: 4x5 one hole:
#       1 left if hole at [1,2] or [2,3]
#       2 left if hole at other places
# Board: 5x5 one hole:
#       3 left if hole at [3,3]
# Standard board:
#       1 left in center hole after 50525 function calls
# European board:
#       2 not successful in 500000 function calls
#       3 successful in 30324 function calls
# Toroidal board: 3x3
#       1 left in corner
# Toroidal board: 4x4
#       1 left in corner
# Toroidal board: 5x5
#       1 left in corner
#       needed 3120 function calls
#       Final play list:
#       [3, 2, 1], [5, 1, 2], [11, 6, 1], [1, 2, 3],
#       [3, 4, 5], [8, 7, 6], [10, 6, 7], [12, 7, 2],
#       [14, 15, 11], [16, 21, 1], [1, 2, 3], [17, 22, 2],
#       [2, 3, 4], [4, 5, 1], [18, 23, 3], [19, 24, 4],
#       [4, 3, 2], [2, 1, 5], [25, 20, 15], [15, 11, 12],
#       [12, 13, 14], [14, 9, 4], [4, 5, 1]]
```