hw5b

B511209 최아성

June 2019

1 hanoi

1.1 hanoi 알고리즘 설계

move(N,S,D,E)
N은 원판 1 부터 N 원판이 쌓인 탑.
S는 N탑의 현재 위치, 출발지.
D는 N탑을 옮기고자 하는 위치.
E는 현재 비어있는 곳(or N원판보다 큰 원판이 있음)
즉 E는 N탑의 모든 원판이 갈 수 있는 곳.

N 탑을 D(목적지)로 옮기기 위해서는 N 원판부터 D로 옮겨야 한다.
1. 그렇기 위해서는 N-1 탑을 먼저 빈공간(E)에 옮겨야한다.
2. N 원판을 D로 옮긴다.
3. 빈공간(E)에 있는 N-1탑을 다시 D의 N원판 위에 옮긴다.

1,3은 move함수가 재귀적으로 실행 할 수 있으며,
move함수의 마지막은 원판이 0일때 아무것도 하지 않고
그 전의 탑인 1탑 = 1원판 이므로 1 원판만 2. 실행문으로 옮기면 된다
실행문 3.에서 다시 N원판에 N-1탑을 쌓을 때는, S D E 위치 모두 N-1원판보다 크므로 move함수는 hanoi 규칙에 어긋나지 않는다.

자세한 것은 코드 주석으로 다루겠습니다.

1.2 hanoi 코드

```
N개의 원판 하노이탑 실행!
hanoi(N) :-
        move(N, 1, 2,3).

//위의 탑(원판)이 더이상 없으면 그냥 true 반환
move(0, _, _, _) :-!.


 //N은 크기가 1 N의 원탑이 모두 쌓인 탑을 의미한다.
 //move는 N을 S(출발지)에서 D(도착지)로 옮긴다.
 //이때 E는 빈공간, 혹은 N보다 큰 원판이 있는 곳
move(N, S, D, E) :-
        N2 is N-1,                  //자신 위의 탑! N2
        move(N2, S, E, D),          //원판 N을 D로 놓기 위해선 탑 N2를
E로 치워 놓아야 한다!
        print(N, S, D),             //이제 원판 N을 목적 D에 옮긴다.!
        move(N2, E, D, S).          //E로 치웠던 탑N2를 다시 D옮겨 자
신의 위에 쌓는다.!

//즉 원판을 옮기기 위해서 위의 탑을 옮겨야 하며
//다시 그 위의 탑을 옮기기 위해서 탑 = 원판 이 되는 순간까지
//재귀적으로 반복하여 실행한다!
//N 원판을 옮겼으면 다시 치웠던 N2 탑을 N원판 위로 쌓아야 한다!
//이때 N2탑은 나머지 자리가 N2탑의 모든원판보다 넓으므로
//move함수를 써서 충분히 다시 N원판 위로 옮길 수 있다.

 //이동 상태 출력 함수
print(N,S,D) :-
        write(N),write("->"),
        write([S,D]), nl.
```

1.3 hanoi trace

중요하지 않은 trace는 생략 하였습니다.

```
Call: (7) hanoi(2) ? creep
Call: (8) move(2, 1, 2, 3) ? creep
——— 2 탑을 무브

————> 1탑을 먼저 빈공간에 옮겨야함
Call: (9) move(1, 1, 3, 2) ? creep

————————————— 1 탑을 무브

————> 0탑을 먼저 빈공간에 옮겨야함
Call: (10) move(0, 1, 2, 3) ? creep
——— 0 탑은 더 이상 옮기지 않고 그냥 true.
Call: (10) print(1, 1, 3) ? creep
—— 1의 원판을 옮김!!
Call: (10) move(0, 2, 3, 1) ? creep
————> 0 탑을 1의 원판위에 쌓아야함
0탑은 그냥 true.
Exit: (9) move(1, 1, 3, 2) ? creep
————————————— 1탑 무비 완료!

(9) print(2, 1, 2) ? creep
———>2 원판을 옮김

이제 1탑을 2원판에 쌓아야함
Call: (9) move(1, 3, 2, 1) ? creep
————> 1 탑을 2원판 위로 옮김
앞에 탑 1을 옮긴 것과 유사!
    Call: (10) move(0, 3, 1, 2) ? creep
    Exit: (10) move(0, 3, 1, 2) ? creep
    Call: (10) print(1, 3, 2) ? creep
    Exit: (10) print(1, 3, 2) ? creep
    Call: (10) move(0, 1, 2, 3) ? creep
    Exit: (10) move(0, 1, 2, 3) ? creep
    Exit: (9) move(1, 3, 2, 1) ? creep
    Exit: (8) move(2, 1, 2, 3) ? creep
    Exit: (7) hanoi(2) ? creep
```

2 Quick Sort

2.1 Quick Sort 알고리즘 설계

quicksort([F |L], SORT)
F는 정렬할 리스트의 첫번째 원소이자, 피벗이고 SORT는 정렬된 리스트의 반환!
qucisort함수는

1.partition()
피벗을 기준으로 리스트를 left(피벗 보다 작은)와 right(피벗 보다 큰)로 나눈다
2.재귀적으로 left와 right리스트를 quicksort한다.
3.정렬된 left와 right를 다시 merge한다.

quicksort는 sort할 배열이 들어오지 않으면 true.로 끝
sort할 리스트 원소가 1개면 Merge만 하고 true.

자세한 것은 코드 주석으로 다루겠습니다.

2.2 Quick Sort 코드

```
quicksort 한 배열 출력!
quickSort(R):- quicksort(R,R1),write(R1),nl.
sort할 배열이 없으면 true.하고 끝.

quicksort([], []).
리스트 원소가 한개 일때는 SORT에 추가 후 merge로 출력
quicksort([F|[]],SORT):-
        write("merge : "),write([F]),nl,
        merge([],[F|[]],SORT).

 F는 맨앞 원소 피벗, L은 나머지 리스트 SORT는 정렬된 리스트
quicksort([F | L], SORT) :-
        F 피벗 기준으로 배열 두개로 나눔
        partition(F, L, LEFT, RIGHT),
        write("divide = "), write(F),write("|"), write(LEFT),write(RIGHT),nl,
        나눠진 배열 재귀적으로 quicksort반복
        quicksort(LEFT, SL),
        quicksort(RIGHT, SR),
        나눠진 배열을 다시 합침!!
        merge(SL, [F| SR], SORT),
        합치면 merge출력
        write("merge : "),write(SL),write([F]),write(SR),nl.


//피벗을 기준으로 리스트를 작은 것과 큰것 두개로 나눠 만든다.
//리스트가 없으면 true하고 끝
```

```
partition(PIVOT, [], [], []).
//피벗을 뺀 리스트에서 F 값이 피벗보다 작으면 LEFT에 저장
partition(PIVOT, [F | L], [F | LEFT], RIGHT) :- F =< PIVOT,
 저장한 원소를 제외하고 나머지 리스트 partition 실행!
 partition(PIVOT, L, LEFT, RIGHT).
//크면 RIGHT에 저장
partition(PIVOT, [F | L], LEFT, [F | RIGHT]) :- F > PIVOT,
 partition(PIVOT, L , LEFT, RIGHT).
```

2.3 Quick Sort trace

중요하지 않은 trace는 생략 하였습니다.

```
[trace]  ?- quickSort([2,3,1]).
Call: (7) quickSort([2, 3, 1]) ? creep 출력 quciSort
진짜 quicksort하는 함수 (sort의 s 소문자..)
Call: (8) quicksort([2, 3, 1], _G1215) ? creep
Call: (9) partition(2, [3, 1], _G1216, _G1217) ? creep
-- 피벗 2로 partition!!
Exit: (10) 3>2 ? creep
Call: (10) partition(2, [1], _G1219, _G1208) ? creep
Exit: (11) 1=<2 ? creep
Call: (11) partition(2, [], _G1211, _G1208) ? creep
Exit: (9) partition(2, [3, 1], [1], [3]) ? creep
-- 재귀하면서 left right로 큰거 작은거 분류
Call: (9) quicksort([1], _G1228) ? creep
Exit: (9) quicksort([1], [1]) ? creep
-- left quicksort,원소가 하나라서 재귀 없음
Call: (9) quicksort([3], _G1238) ? creep
Exit: (9) quicksort([3], [3]) ? creep
 -- right quicksort,원소가 하나라서 재귀 없음
 Call: (9) backward_compatibility:merge([1], [2, 3], _G1252) ? creep
 -- merge, pivot은 right에 껴서 같이 merge ! 출력할때는 구분됨
  Exit: (8) quicksort([2, 3, 1], [1, 2, 3]) ? creep
   -- sorting한거 반환해서 줌!
   Call: (8) write([1, 2, 3]) ? creep
[1,2,3]
   Exit: (8) write([1, 2, 3]) ? creep
   Call: (8) nl ? creep
   Exit: (8) nl ? creep
   Exit: (7) quickSort([2, 3, 1]) ? creep
true .
```

3 nQueen

3.1 nQueen 알고리즘 설계

queen([Q | Qlist],N)
각각 N개의 행마다 nQueen문제에 적합한 col을 선택하는 함수

Q는 현재 행에서 선택할 col !
Qlist는 나머지 행 리스트, N은 총 열의 개수
queen함수는 재귀적으로 queen(Qlist,N)가 true여야 한다.
queen(Qlist,N)이 true라는 것은 Q를 이전의 행은 nQueen에 맞춰 놓아졌다는 것!!
queen()는 Qlist가 비었으면 그냥 true.

1.그 전의 행이 모두 nQueen에 맞춰 놓아졌다면,
2.Q는 1 부터 N 까지의 col중에 promising한 col을 선택한다
3.promising하다는 것은
1) 같은 열이 없다는 것
2) 대각선상에 놓여있지 않다는것 즉 (행차이 != 열차이)

자세한 것은 코드 주석으로 다루겠습니다.

3.2 nQueen 코드

```
nQueen(N) :-
        //길이 N 배열 Solution 생성
        length(Solution, N),
        write('n='),write(N),nl,
        //L 리스트에 모든 해답 저장
        findall(Solution,queen(Solution, N),L),
        length(L,LEN),
        write("# of answer="),write(LEN),nl,
        write(L),nl.

//k 부터 N까지 수를 가진 리스트 생성
//K를 1씩 증가하며 QLIST에 추가하는 재귀 함수
qlist(N,N,[N]) :-!.
qlist(K,N,[K|CLIST]) :- K < N, K1 is K+1, qlist(K1, N, CLIST).

//queen을 돌려볼 배열이 비었으면 true
queen([],_).
```

```
//Qlist에 Q(col)을 순서대로 입력
queen([Q|Qlist],N) :-

    //나머지, Q를 제외한 N-1개 원소, Qlist에 재귀적으로 실행
        queen(Qlist, N),
    //Qlist까지 true하게 nQueen이 놓아졌다면 Q만 잘 놓아지면 된다!

        //N개 퀸은 위치 할 수 있는 cal 후보가 1 부터 N, 리스트 생성
        qlist(1,N,Can),

        //1 부터 N에 들어있는 true로 반환되는 Q가 후보다.!
        member(Q, Can),

        //그리고 그 Q는 Qlist에 추가할때 promising도 true 해야됨!
        promising(Q,Qlist, 1).


//promising할 리스트가 비었으면 true
promising(_,[], _).

//Q는 현재 놓아보는 col 후보자! Q1은 이미 놓아져서 비교할 col!
promising(Q,[Q1|Qlist],Srow) :-
        Q =\= Q1,       //같은 col 인지 검사!!
        Test is abs(Q1-Q),          //절대값으로 저장 Q1-Q
    //Srow는 행의 차이,즉 처음은 1, 행차이 = 열차이는 대각선에 위치를 뜻함
        Test =\= Srow,                          //행차이 != 열차이
        Srow1 is Srow + 1,                      //다음행 검사를 위한 +1
        promising(Q,Qlist,Srow1).               //다음 행들도 promising
한지 검사!
```

3.3 nQueen trace

중요하지 않은 trace는 생략 하였습니다.

```
[trace]  ?- nQueen(1).
   Call: (7) nQueen(1) ? creep
   Call: (8) length(_G1196, 1) ? creep
   nQueen(1) 실행 리스트 크기 1배정
^  Call: (8) findall([_G1189], queen([_G1189], 1), _G1204) ? creep
    모든 true findall로 리스트 취합!
   Call: (13) queen([_G1189], 1) ? creep
   queen 실행!, promissing한 col찾기!
   Call: (14) queen([], 1) ? creep
   그 다음행 먼저 실행하러 옴!!
   dfs처럼 쭉 들어가서 첫 행부터 promissing하게 놓음
   Exit: (14) queen([], 1) ? creep
   빈 리스트를 만나면 탈출!!
   n=1이다 보니 바로 다음 행이 빈리스트...
   Exit: (14) qlist(1, 1, [1]) ? creep
   Call: (14) lists:member(_G1189, [1]) ? creep
   Exit: (14) lists:member(1, [1]) ? creep
   놓을 수 있는 n =1개의 col 후보
   Call: (14) promising(1, [], 1) ? creep
   promising이 n부터 1까지 행들을 검사함
   [1 n-1]리스트에서 하나씩 꺼내는 건데 n=1이라 빈리스트
   Exit: (14) promising(1, [], 1) ? creep
   n=1이라서 한번 검사하고 끝나버림

   Exit: (13) queen([1], 1) ? creep
   queen 정답 하나 찾음!

   Redo: (14) promising(1, [], 1) ? creep
   Fail: (14) promising(1, [], 1) ? creep
   Fail: (13) queen([_G1189], 1) ? creep
    true인 것을 더 찾지 못함
^  Exit: (8) findall([_G1189], user:queen([_G1189], 1), [[1]]) ? creep
   Call: (8) length([[1]], _G1226) ? creep
   Exit: (8) length([[1]], 1) ? creep
 정답 [[1]] 한개로 끝
```

4 Shortest Path

4.1 Shortest Path 알고리즘 설계

주어진 길들을 p(출발지,도착지,비용) 형태로
true. 룰을 정의한다
path() 함수는 출발지와 도착지를 받아 모든 경로와 비용을 구한다.

1. 출발지에서 바로 연결되어 있는 true한 길(mid)을 구한다.
2. 그 길이 지금까지 경유한 노드가 아니어야 한다. not member(Mid, [S — CPath])
3. 바로 연결한 길의 비용을 지금까지의 비용에 더한다.
4. 출발지를 지금 경유한 mid로 하여 path함수를 재귀실행한다
5. mid가 도착지가 될때까지 실행한다.

이렇게 구한 모든 경로들 중에서 비용이 가장 작은 것을 선택한다

자세한 것은 코드 주석으로 다루겠습니다.

4.2 Shortest Path 코드

```
//path 지정!
//p(출발점, 도착점, 비용)
//이 path들은 모두 true
p(1,2,6). p(1,3,3). p(2, 1, 6).
p(2,3,2). p(2,4,5). p(3,1,3).
p(3,2,2). p(3,4,3). p(3,5,4).
p(4,2,5). p(4,3,3). p(4,5,2).
p(4,6,3). p(5,3,4). p(5,4,2).
p(5,6,5). p(6,4,3). p(6,5,5).

//도착했을때 true, (세번째 인자)에 길을 반환
path([H|R], H, [H|R], Length, Length).

//S 에서 D로 가는 모든길
//CPath는 현재까지 가고있는 길
//CL은 현재까지 가는데 모은 비용, L은 총 길이 비용 반환값

path([S | CPath ], D, ResltPath, CL, L) :-
        p(S, Mid, X),                    //S에서 갈 수 있는 true 길!
        \+member(Mid, [S | CPath]), //이미 지났던 길은 제외!
        NewL is CL + X,                        //비용 X 추가 !!

        //Mid를 CPath에 추가, Mid부터 갈수 있는길 조사!
        path( [ Mid, S | CPath ] ,D , ResltPath ,NewL,L).
```

```
sp(S, D) :-
        //S 에서 D로 모든 경로 SET에 저장
        findall([L, R ],path([S], D, R, 0, L),SET),
        //비용순으로 정렬후 가장작은 첫원소 비용과 경로 LEN,PATH
        sort(SET,[ [ LEN,PATH | DP ]|SORT ]),
        reverse(PATH,DPATH),
        //경로 추가시 앞에서부터 추가했으므로 다시 역순으로!

        //경로, 길이 출력!
        write(DPATH),nl,
        write(LEN),nl.
```

4.3 Shortest Path trace

중요하지 않은 trace는 생략 하였습니다.

```
Call: (7) sp(1, 3) ? creep
1 -> 3 모든 경로를 다 구해본다.
Call: (8) findall([_G1191, _G1194], path([1], 3, _G1194, 0, _G1191), _G1215)
Call: (13) path([1], 3, _G1194, 0, _G1191) ? creep
Call: (14) p(1, _G1231, _G1232) ? creep
Exit: (14) p(1, 2, 6) ? creep
-- p(1,mid,x) true인거중 하나
Call: (14) lists:member(2, [1]) ? creep
-- [1]은 지금까지 경유한 노드 2는 경유한적 없는 노드여야 한다. true이다.
Exit: (14) 6 is 0+6 ? creep
Call: (14) path([2, 1], 3, _G1194, 6, _G1191) ? creep
-- 비용 +6, 경유 노드에 2를 추가한후 출발지를 2로 선택후 path 재귀실행
Call: (15) p(2, _G1240, _G1241) ? creep
Exit: (15) p(2, 1, 6) ? creep
Call: (15) lists:member(1, [2, 1]) ? creep
Exit: (15) lists:member(1, [2, 1]) ? creep
-- p(2,1,6)선택, mid = 1, 경유 노드에 걸려서 탈락
Exit: (15) p(2, 3, 2) ? creep
Call: (15) lists:member(3, [2, 1]) ? creep
p(2,3,2)는 경유 가능한 선택!!
Exit: (15) 8 is 6+2 ? creep
Call: (15) path([3, 2, 1], 3, _G1194, 8, _G1191) ? creep
Exit: (15) path([3, 2, 1], 3, [3, 2, 1], 8, 8) ? creep
출발지 mid = 도착지면 경로와 비용을 반환하고 true

같은 방법으로 나머지 true가능한 경로들을 findall로 전부 찾음
비용을 기준으로 sorting하여, 첫번째(가장작은 비용)의 경로를 선택 출력!
```