<전체 알고리즘 코드>

```
function ITERATIVE-LENGTHENING-SEARCH (problem) returns a solution or failure
1   limitcost ← 0
2   node ← MAKE-NODE( problem.INITIAL-STATE )
3   while ( limitcost < ∞)
4       (result, min_cost, reachgoalnode) ← MY_DLS(node, problem, limitcost)
5       if result == success then return SOLUTION(reachgoalnode)
6       else limitcost ← min_cost
7   return failure

function MY_DLS(node, problem, limitcost) return ( Result, Cost, Reachnode )
1    Info_list ← [ ( cutoff, ∞, node ) ]
2    for each action in problem.ACTIONS( node.STATE ) do
3        child ← CHILD-NODE( problem, node, action )
4        (result, cost, reachnode) ← LIMITCOST_CHECK(child, problem, limitcost)
5        Info_list.APPEND( result, cost, reachnode )

6    mincost ← MIN( Info_list[1] )
7    if success in Info_list[0] then i ← INDEX( Info_list[0] == success )
8    return ( Info_list[0][i], Info_list[1][i], Info_list[2][i] )
9    else j ← LAST-INDEX( Info_list ) return ( Info_list[0][j], mincost, Info_list[2][j] )

function LIMITCOST_CHECK (node, problem, limitcost) return ( result, cost, reachnode )
1     if PATH-COST(node) > limitcost AND
2     problem.GOAL-TEST(node.STATE) == true then
3         return ( not_guaranteed_optimalsolution, PATH-COST(node), node )

4     else if PATH-COST(node) > limitcost AND
5     problem.GOAL-TEST(node.STATE) == false then
6         return ( cutoff, PATH-COST(node), node )

7     else if PATH-COST(node) <= limitcost AND
8     problem.GOAL-TEST(node.STATE) == true then
9         return ( success, PATH-COST(node), node )

10    else if PATH-COST(node) <= limitcost AND
11    problem.GOAL-TEST(node.STATE) == false then
12      action ← problem.ACTIONS( node.STATE )
13      nextnode ← CHILD-NODE(problem, node, action)
14      LIMITCOST_CHECK (nextnode, problem, limitcost)
```

<전체 알고리즘에 대한 설명>

ITERATIVE-LENGTHENING-SEARCH를 구현하는데에 근본적인 아이디어는 DLS에서 depth를 하나씩 늘리는 대신 출발 node에서 어떤 한 node까지의 PATHCOST를 구한 후, 그 값들중 가장 작은 값(lowest path cost)을 다음 limit으로 정하여, **loop**를 도는 형식으로 결국 PATHCOST가 점진적으로 증가하는 **UNIFORM-COST-SEARCH** 와 동일한 결과를 얻지만, **DEPTH-FIRST_SEARCH**기반과 **ITERATIVE DEEPING**원리 이용하고, *explored*를 전혀 사용하지 않고, *frontier*(chlid-node)만 Memory Space에 있으면 되므로, Worst Case인 경우에 대해 Space Complexity를 모두 계산하면 결국, $O(\ b + (b-1)*(d-1)\ ) \Rightarrow O(bd)$ 입니다. 따라서 space면에서 복잡도가 기존의 **UNIFORM-COST-SEARCH**에 비해 낮아지는 장점이 있는 알고리즘입니다. 제가 구현한 **함수**는 **ITERATIVE-LENGTHENING-SEARCH**, **MY_DLS**, **LIMITCOST_CHECK** **3**가지 입니다. 각 함수에 대한 구체적 설명은 아래에서 설명하겠습니다.

---

function **ITERATIVE-LENGTHENING-SEARCH** (*problem*) **returns** a solution or failure

1 *limitcost* ← 0
2 *node* ← MAKE-NODE(*problem*.INITIAL-STATE)
3 **while** (*limitcost* < ∞)
4 (*result, min_cost, reachgoalnode)* ← **MY_DLS**(*node*, *problem*, *limitcost*)
5 **if** *result* == *success* **then return** SOLUTION(*reachgoalnode*)
6 **else** *limitcost* ← *min_cost*
7 **return** *failure*

---

1. 처음에는 *limitcost*을 0으로 설정합니다. (첫 루프를 위함)
2. *node*를 MAKE-NODE(*problem*.INITIAL-STATE)를 통해 root 노드로 설정합니다.

3.4.5.6. **loop**를 도는데, **MY_DLS**은 (Result, Cost, Reachnode) 인 튜플 자료구조를 **return**합니다. 이때, *result*는 *not_guaranteed_solution, cutoff, successf* 중 하나이고, *min_cost*는 **LIMITCOST_CHECK**에서 구한 PATH-COST중 가장 작은 값(lowest path cost) 가 됩니다. 만약 *result*가 *success* 라면, SOLUTION(*reachgoalnode*)로 solution을 **return** 합니다. *success*가 아니라면, *limitcost*를 *min_cost*로 업데이트 시켜서 다음 **loop**를 돌게됩니다.

7. 만약 모든 경우를 탐색한 경우라면 *node*에 *child*가 없는 경우라면, 반복문을 빠져나오고 *failure*를 **return** 합니다.

function **MY_DLS**(*node, problem, limitcost*) **return** (Result, Cost, Reachnode)

1 *Info_list* ← [(*cutoff,* ∞, *node*)]
2 **for each** *action* in *problem*.ACTIONS(*node*.STATE) **do**
3 *child* ← CHILD-NODE(*problem*, *node*, *action*)
4 (*result*, *cost*, *reachnode*) ← **LIMITCOST_CHECK**(*child, problem, limitcost*)
5 *Info_list*.APPEND(*result, cost, reachnode*)

6 *mincost* ← MIN(*Info_list*[1])
7 **if** *success* **in** *Info_list*[0] **then** i ← INDEX(*Info_list*[0] == *success*)
8 **return** (*Info_list*[0][i], *Info_list*[1][i], *Info_list*[2][i])
9 **else** j ← LAST-INDEX(*Info_list*) **return** (*Info_list*[0][j], *mincost*, *Info_list*[2][j])

---

1. *Info_list* 라는 이름의 tuple이 원소인 list형 자료구조를 생성하고, *node*에 *child*가 없는 최종 마지막 탐색의 경우를 염두해서 (*cutoff,* ∞, *node*)로 list를 초기화 합니다.

2.3.4.5. *problem*.ACTIONS(*node*.STATE)로 *action*을 구하고 그 *action*에 따라 CHILD-NODE(*problem*, *node*, *action*)을 통해 *node*의 *child*를 구합니다. 그리고 **LIMITCOST_CHECK**(*child, problem, limitcost*) 함수를 호출합니다. **LIMITCOST_CHECK**의 기능은 아래에서 설명하겠습니다. 어쨌든, **LIMITCOST_CHECK** 함수는 (result, cost, reachnode)를 **return** 합니다. 그리고 **return**된 (result, cost, reachnode) 값을 *Info_list*에 추가(APPNED)합니다. 각 *action*에 대해 반복문이 끝나면,

6. MIN(*Info_list*[1])를 통해서 *cost*의 여러값 중에 가장 작은 값(lowest path cost)을 *mincost*에 할당합니다.

7.8. 그리고 if문을 통해 *Info_list*[0]에 success가 있으면, 그때의 *index*를 INDEX(*Info_list*[0] == *success*)를 통해 I에 할당하고, *Info_list*에서 i값에 해당하는 tuple (*Info_list*[0][i], *Info_list*[1][i], *Info_list*[2][i])값을 **return** 합니다.

9. 만약 *success*가 *Info_list*[0]에 존재하지 않는다면, *Info_list*의 마지막 index를 LAST-INDEX(*Info_list*)을 통해 j에 할당하고, (*Info_list*[0][j], mincost, *Info_list*[2][j])를 **return** 합니다.

```
function LIMITCOST_CHECK (node, problem, limitcost) return ( result, cost, reachnode )
1     if PATH-COST(node) > limitcost AND
2     problem.GOAL-TEST(node.STATE) == true then
3        return ( not_guaranteed_optimalsolution, PATH-COST(node), node )

4     else if PATH-COST(node) > limitcost AND
5     problem.GOAL-TEST(node.STATE) == false then
6        return ( cutoff, PATH-COST(node), node )

7     else if PATH-COST(node) <= limitcost AND
8     problem.GOAL-TEST(node.STATE) == true then
9        return ( success, PATH-COST(node), node )

10    else if PATH-COST(node) <= limitcost AND
11    problem.GOAL-TEST(node.STATE) == false then
12      action ← problem.ACTIONS( node.STATE )
13      nextnode ← CHILD-NODE(problem, node, action)
14      LIMITCOST_CHECK (nextnode, problem, limitcost)
```

1.2.3. 첫 번째 if 문은 PATH-COST(*node*) > *limitcost*이고 *node*가 목적지(goal) 에 도착한 경우입니다. 이 경우는 목적지(goal)에는 도착 하였지만, PATH-COST(*node*) > *limitcost*임으로 optimal한 solution이라고 보장 할 수 없습니다. 따라서 (*not_guaranteed_optimalsolution*, PATH-COST(*node*), *node*)인 tuple을 **return** 합니다.

4.5.6. 두 번째 if 문은 PATH-COST(*node*) > *limitcost*이고 *node*가 목적지(goal) 에 도착하지 않은 경우입니다. node가 goal에 도착하지도 않았고, PATH-COST(*node*) > *limitcost* 임으로, *result*가 *cutoff*인 상태입니다. 따라서 (*cutoff,* PATH-COST(*node*), *node*)인 tuple을 **return** 합니다.

7.8.9. 세 번째 if 문은 PATH-COST(*node*) <= *limitcost*이고 *node*가 목적지(goal) 에 도착한 경우입니다. 이 경우는 goal을 찾고 PATH-COST(*node*) <= *limitcost*임으로, optimal한 solution을 찾은 경우(*result* == *success*)입니다.
따라서 (*success,* PATH-COST(*node*), *node*)인 tuple을 **return** 합니다.

10.11.12.13.14. 네 번째 if 문은 PATH-COST(*node*) <= *limitcost*이고 *node*가 goal에 도착하지 않은 경우입니다. 이 경우에는 *action* ← *problem*.ACTIONS(*node*.STATE)와 *nextnode* ← CHILD-NODE(*problem*, *node*, *action*)로 현재 *node*에 대한 *nextnode(현재 node*에 대한 *child)*를 구한 후에, 재귀적 으로 *nextnode*를 **LIMITCOST_CHECK** 함수의 첫 번째 인자(*node*)로 넣게 되면, **그 함수**는 첫 번째, 두 번째, 세 번째 조건문에 걸릴 게 되고 함수는 종료됩니다.