Iterative Lengthening Search 알고리즘을 이용한 모바일 퍼즐 게임 <RGB Express> Solver 제작

https://github.com/doobee98

1. 요약

이번 과제물에서는 일반적으로 적용될 수 있는 Iterative Lengthening Search 알고리즘을 Python으로 구현하고, 그 알고리즘을 이용하여 모바일 퍼즐 게임 <RGB Express>의 Solver를 제작한다. 문제 그림을 어느 정도 문자 및 데이터 형태로 변환하여 테스트케이스로 넣으면, 해당 알고리즘이 이해할 수 있는 Problem으로 변환해 주는 간단한 Parser를 포함한다.

이번 과제의 목적은 다음과 같다.

1. Space Complexity가 O(bd)인 Iterative Lengthening Search 알고리즘을 구현할 것.
2. 해당 알고리즘의 수행 과정을 창의적인 예제를 이용하여 그림에서 설명할 것.

2. Iterative Lengthening Search 알고리즘 구현

(myPriorityQueue.py, problem.py, search.py)

Space Complexity가 O(bd)인 Iterative Lengthening Search 알고리즘을 구현할 것

언어의 차이에 따라 구현 이슈가 발생하지는 않을 것 같아서, 개인적으로 익숙한 언어인 Python을 선택하였다. 구현을 위해서는 먼저 문제(Problem)을 형식화해야 한다.

```python
class Problem:
    def __init__(self,
                 initial_state: State,
                 actions: Callable[[State], List[Action]],
                 transition_model: Callable[[State, Action], State],
                 goal_test: Callable[[State], bool],
                 path_cost: Callable[[State, Action], int]):
        self.__initial_state = initial_state
        self.__actions = actions
        self.__transition_model = transition_model
        self.__goal_test = goal_test
        self.__path_cost = path_cost

    """
    problem methods
    * initialState
    * actions
    * doAction
    * isGoalState
    * pathCost
    """
```

문제는 다섯 가지 요소로 구성된다. Initial State는 State중 하나로 문제의 초기 상태를 의미한다. Actions는 State를 입력으로 받아서, 가능한 Action들의 리스트를 반환해주는 함수이다. Transition Model은 State와 Action을 입력으로 받아 다음 State를 반환하는 함수이다. Goat Test는 State를 입력으로 받아 현재 State가 Goal인지 아닌지 판단하여 bool 값으로 반환하는 함수이다. 마지막으로 Path Cost는 State와 Action을 입력으로 받아 해당 Action의 Cost를 반환하는 함수이다.

이어서 MyPriorityQueue를 구현하였다. Python에서는 queue 내장 모듈에서 PriorityQueue 클래스를 사용할 수 있지만, 해당 클래스에서는 구현할 알고리즘에서 필요한 findItem()과 remove(item) 기능의 사용이 어려웠기 때문에 heapq를 이용하여 MyPriorityQueue를 직접 구현하여 사용하였다.

그 후에는 Uniform Cost Search 알고리즘을 구현하였다(결과 비교를 위해서). Uniform Cost Search 알고리즘은 강의자료 2(Blind Search 1) 29p의 수도코드를 그대로 구현하였다. 구현 이슈로, Priority Queue가 담는 객체가 Node(Tuple[int, State, List[Action]] – Tuple of cost, state, action list)이기 때문에 State의 __lt__ 매직메소드를 구현하여야 했으며, 노드의 크기 비교는 cost로만 비교해도 되기 때문에 비교 함수는 단순히 True를 반환하도록 하였다.

```
"""
    HomeWork Algorithm: Iterative Lengthening Search
"""
TypeSuccess = Tuple[int, List[Action]]  # Best Cost, Solution Path의 튜플
TypeFailure = type(None)
TypeCutoff = int                        # Cutoff된 Cost중 최솟값을 저장


def iterative_lengthening_search(problem: Problem, show_log: bool = False) -> Optional[List[Action]]:
    # recursive_CLS 함수는 주어진 cost limit에 대해 Success Tuple, Cutoff value, Failure 셋 중 하나를 반환한다.
    def recursive_CLS(state: State, cost: int, cost_limit: int) -> Union[TypeSuccess, TypeCutoff, TypeFailure]:
        if problem.isGoalState(state):
            return cost, []
        if cost > cost_limit:
            return cost

        cutoff_value = None     # Goal State나 Cutoff가 발생하지 않으면 그대로 Failure인 None이 되도록 초기값 설정
        for action_iter in problem.actions(state):
            next_state = problem.doAction(state, action_iter)
            next_cost = cost + problem.pathCost(state, action_iter)
            result = recursive_CLS(next_state, next_cost, cost_limit)
            # Cutoff시 Cutoff당한 cost중 최솟값을 cutoff_value에 저장한다.
            if isinstance(result, TypeCutoff):
                if cutoff_value is None or result < cutoff_value:
                    cutoff_value = result
            # Failure시 이 state는 무시하고 다음 state를 탐색함
            elif isinstance(result, TypeFailure):
                continue
            # 둘 다 아니라면 Solution Path를 더해줌
            else:
                cost, action_list = result
                return cost, [action_iter.value()] + action_list
        return cutoff_value

    cost_limit = 0
    while True:
        if show_log:
            print('\tFind Cost:', cost_limit)
        result = recursive_CLS(problem.initialState(), 0, cost_limit)
        # Cutoff될 시 다음 Cost Limit은 Cutoff된 최소의 Cost를 이용하여 재탐색 (cost_limit이 증가함)
        if isinstance(result, TypeCutoff):
            cost_limit = result
        # recursive_CLS의 결과가 None일시 가능한 Path가 없으므로 None 반환
        elif isinstance(result, TypeFailure):
            return None
        # 탐색에 성공했을 경우 최적 코스트를 출력 후 해당 액션 리스트를 반환함
        else:
            cost, action_list = result
            print('< Optimal Cost', cost, '>')
            return action_list
```

마지막으로 목표 알고리즘인 Iterative Lengthening Search 알고리즘을 구현하였다. 탐색에 성공할 경우 Solution Path를 반환하고, 실패한다면 Failure의 의미로 None을 반환한다. 내부의 recursive_CLS 함수는 explored나 frontier를 사용하지 않고 DFS처럼 recursive하게 하나의 분기만 탐색하기 때문에 Space Complexity는 O(bd)이다(limit가 depth가 아닌 cost라는 점이 Depth Limited Search와의 차이점이다 – Cost Limited Search). 탐색에 성공할 경우 Best Cost와 Solution Path를 반환하며, 탐색에 실패한 경우는 두 가지로 나뉜다. 더 이상 탐색할 것이 없는 경우는 Failure를 반환하며, 탐색할 것이 있지만 cost limit에 걸린 경우에는 Cutoff를 반환한다. Cutoff된 경우 다음 Iterative 탐색에서 Cost Limit를 증가시켜야 하는데, 이전 탐색에서

Cutoff된 Cost중 가장 작은 값으로 증가시키면 유한한 상태 집합에 대해 알고리즘의 Completeness와 Optimality를 보장할 수 있다(Best Cost는 해당 cost보다 낮은 cost를 가진 모든 가능한 Path를 탐색한 후 도출된 값이므로). 따라서 해당 정보를 전달하기 위해 recursive_CLS에서 반환하는 TypeCutoff는 int이며, 해당 값은 Cutoff된 Cost 중 최솟값이다. iterative_lengthening_search 함수에서는 recursive_CLS 함수에서 TypeFailure나 TypeSuccess이 반환되면 None이나 Solution Path를 반환하며, TypeCutoff가 반환되면 상기에 기술한 대로 cost limit를 증가시키면서 iterative하게 recursive_CLS 함수를 호출한다.

여기까지 하여 General한 Uniform Cost Search 알고리즘과 Iterative Lengthening Search 알고리즘을 Python으로 구현하였으며, 강의 도중 예시로 나온 <Route Finding Problem>과 <8 Queens Problem>을 예시로 작성하여 테스트하였다. 해당 문제들의 구현은 소스 코드 중 toyProblem 폴더에서 확인할 수 있다. 탐색 결과를 확인할 때, Cost가 같은 경우 세부적인 Solution Path는 약간 다를 수 있음에 유의한다.

3. 목표 문제 설정

Iterative Lengthening Search은 비용이 다른 Weighted Graph에서 Optimal Solution을 찾는 데에 적합한 알고리즘이다. 비용이 다르지 않은 경우 Iterative Deepening Search의 효율이 더 좋다. 따라서, 목표 문제를 설정하는 데에 있어 다음과 같은 원칙을 정했다.

(a) 항상 Action의 Cost가 같은 경우를 배제한다. (ex – 8-Queens)

(b) 해당 알고리즘을 적용할 수 있는 문제여야 한다. (ex – 무한한 상태집합)

비용이 다를 때 최소 비용을 탐색하는 경우이므로, 최소 시간, 최소 거리, 최소 비용 등을 탐색하는 데에 적합하다. 실생활 예시로는 지하철 노선도 길찾기와 같은 경우를 예로 들 수 있는데, 창의적이라고 보기는 어려우므로 다른 문제를 찾아보기로 결정하였다.

평소 취미로 보드게임이나 모바일 퍼즐 게임을 즐겨 하는 편인데, 지금까지 해 보았던 1인용 게임 중에서 적합한 문제를 찾은 후 해당 퍼즐이나 문제를 푸는 Solver를 만들어 보면 좋겠다는 생각이 들었다. 이전에 했던 게임들 중에서 Cost가 다른 경우를 찾아보았고, 결과적으로 <RGB-Express>라는 모바일 퍼즐 게임을 선택하였다. 이 게임은 길의 길이가 제각각 이기 때문에 Action마다 Cost가 같기 어렵고, 한붓그리기의 특성상 상태집합 역시 유한하다.

4. RGB-Express 게임 소개

https://play.google.com/store/apps/details?id=com.badcrane.rgbexpress&hl=ko

1인용 모바일 퍼즐 게임 <RGB-Express>의 룰에 대해서 간단하게 알아보려고 한다. 여러 색깔의 트럭, 화물, 창고가 있으며, 모든 화물을 창고에 배달하는 것을 목표로 하는 게임이다. 예를 들면, 빨간 색 화물은 빨간 색 창고로 배달되어야 하며, 빨간 색 트럭으로만 배달할 수 있다. 이번 과제에서는 과제가 너무 복잡해질 것을 우려하여 D단계 이후부터 점점 등장하는 특수 이벤트(무색 트럭, 다리와 다리 스위치, 수륙양용차량 등)는 다루지 않는다. 만일 해당 특수 이벤트들까지 확장해서 다루고 싶다면, RGBEvent나 Truck 등에서 추가하여 다룰 수 있다.

기본적인 세부 규칙은 다음과 같다.

1. 트럭은 등속 운행한다.
2. 한 트럭에는 최대 3개의 화물을 실을 수 있다.
3. **(중요)** 교차로는 여러 트럭이 지날 수 있지만, 한번 트럭이 지나간 교차로가 아닌 길은 해당 트럭이나 다른 트럭이 더 이상 지나갈 수 없다. 한붓그리기 문제와 비슷하다.
4. 여러 트럭들이 같은 지점을 동시에 지난다면 충돌하고, 즉시 실패한다.
5. 트럭은 한번 멈추면 더 이상 움직일 수 없다.
6. 트럭이 다른 색상의 창고에 도달할 경우 바로 실패한다.
7. 창고는 교차로가 아닌 길 1곳에서 상호작용할 수 있다.

게임 내에서 <비용>을 정확히 정의하고 있지 않아서 임의로 정의해야 했다. 이번 과제에서 특정 <RGB-Express>문제의 비용은 "Goal State에 도달할 때까지 등속 운행하는 트럭이 각각 이동한 거리의 최댓값, 즉 Initial State부터 Goal State에 도달하기까지 걸린 시간"을 비용으로 정의한다. 특정 Action의 비용은 5. 문제 형식화에서 정의한다.

5. 문제 형식화 (mainProblem/rgbexpress.py)

2에서 만든 알고리즘으로 <RGB-Express> 문제를 풀기 위해서는 해당 문제를 Problem 클래스로 표현하는 문제 형식화 과정을 거쳐야 한다. 해당 파일에서 구현 한 내용이 다소 많기에 상세 내용의 설명은 주석으로 대신하고, 이 파트에서는 추가설명이 필요한 부분을 서술하려고 한다.

- rgbTrans: 프로그램의 전체적인 논리 흐름을 설명하기 위해 먼저 기술한다. RGBState와 RGBAction을 받아 다음 RGBState를 반환하는 함수이다. 먼저 출발 전 해당 칸에 화물이나 창고가 있다면 화물을 싣거나 내린다. 그리고 RGBAction에 있는 모든 move를 트럭에 적용시키며, 정지하는 것이 아니라면 출발지의 교차로 정보와 도착지의 교차로 정보를 편집하여 앞으로 갈 길을 다른 트럭이 더 이상 진입할 수 없도록 막는다(하단의 RGBEvent 참조). 그리고 앞으로 움직일 트럭들의 remain중 최솟값(하단의 rgbActionCost 참조)인 min_remain을 찾아 트럭들을 min_remain만큼 움직인다. 이 모든 과정을 수행한 후에 new_state를 반환한다.

- RGBEvent: 특정 지점이 교차로이거나, 트럭의 출발지이거나, 화물이 놓여 있거나, 창고가 있는 경우에는 모두 RGBEvent 객체의 관리를 받는다. 여기에서는 이 중에서 교차로 Event에 관해 추가로 설명한다. 한붓그리기와 같이, Event와 Event 사이 Edge(Road)는 모든 트럭을 통틀어서 단 한 번만 지날 수 있다. Edge는 Event와 Event 사이에서 특정 Dir과 Opponent Dir(반대 방향 – Left와 Right같은)의 쌍으로 표시되므로, Event에서 Dir을 Key로, Count(해당 Dir에 있는 다음 Event까지의 거리)를 Value로 하는 Dictionary로 교차로를 관리하자는 것이 메인 아이디어이다. 이 방식에서 한붓그리기는, Event에서 다음 Event로 움직일 때 출발지 Event와 도착지 Event에 있는 각각의 교차로 Dictionary에서 해당 Edge에 해당하는 Dir 정보를 삭제하는 방법으로 구현된다.

  예를 들면, (1,0) Event의 교차로 Dictionary가 {Dir.Right – (0, 1): 5, Dir.Up – (-1, 0): 3}이고, 이 정보를 참조하여 Dir.Right로 움직이기로 결정하였다면, (1, 0) Event의 교차로에서 Dir.Right 정보를 삭제하고, 다음 목적지인 (1, 5) Event의 교차로에서 Dir.Left

정보를 삭제한다.

- RGBState: 트럭 리스트와 event 지도로 표현된다. 이 중 event 지도는 RGBEvent나 None을 담는 이중 리스트이다. RGBState의 event 메소드에 좌표를 인자로 전달하여 접근할 수 있다.
- rgbActions: 특정 RGBState에서 가능한 RGBAction들의 리스트를 계산한다. 여기에서 RGBAction은 해당 state에서 움직일 수 있는 트럭들의 가능한 move를 모은 튜플들의 리스트가 된다. 예를 들어, 현재 state에서 트럭 1은 Up으로 3만큼, 트럭 2는 Right 4만큼 움직인다면 이때의 RGBAction은 [(1, Dir.Up, 3), (2, Dir.Right, 4)]를 의미한다.
- rgbActionCost: RGBAction의 비용은, 앞으로 움직일 트럭들의 remain중 최솟값을 의미한다. Remain이 1인 트럭과 2인 트럭이 있다면, 전자는 1만큼의 거리를 이동하고 Action을 다시 결정해 주어야 하지만 후자는 해당 시점에서는 아직 이동 중이기 때문에 Action을 결정할 필요 없이 remain을 1로 바꾸기만 하면 되기 때문이다.

6. 문제 입력 파싱 (mainProblem/parse.py, mainProblem/testcase.py)

문제를 보다 편리하게 입력하기 위해 기본적인 Parser가 필요했다. Parser의 상세한 구현 설명은 과제의 범위에 포함되지 않으므로 본 과제물에서는 다루지 않으며, parseIntialState 함수의 사용법, 즉 입출력이 무엇인지 기술한다.

```
'A-9': {
    # road tokens: '-', '-', '|', '┐', '└', '|', '
    'Roads': "┐     ┐\n"  # 0
             "├┬┬┬┤\n"  # 1
             "| | | |\n"  # 2
             "├┤ ├┤\n"  # 3
             "| ├┤ |\n"  # 4
             "├┤ ├┤\n"  # 5
             "| | | |\n"  # 6
             "├┴┴┴┤\n"  # 7
             "└     ┘",  # 8
    'Trucks': [
        Truck(0, Color.Red, Dir.Up, (8, 0)),
        Truck(1, Color.Yellow, Dir.Up, (8, 6))
    ],
    'Items': {
        Color.Red: [(3, 2), (5, 2)],
        Color.Yellow: [(5, 4)],
    },
    'Cargos': {
        Color.Red: [(0, 0), (1, 3)],
        Color.Yellow: [(0, 6)],
    }
},
```

(a) 출력: 주어진 입력을 파싱하여 얻은 Initial State를 5에서 형식화한 RGBState(트럭 리스트와 event 지도로 표현됨)의 형태로 출력한다.

(b) 입력: 오른쪽 사진과 같이 표현된다. "Roads", "Trucks", "Items", "Cargos"를 Key로 포함하는 Dictionary의 형태를 띈다. Roads는 개행문자를 구분자로 하여 표현되는 문자열이며, Trucks는 초기 위치와 방향이 지정된 트럭 리스트이다. Items와 Cargos는 각각의 색상의 화물과 창고의 위치 리스트를 보관하는 Dictionary로 저장한다. 트럭 리스트의 경우 생성자에 처음으로 전달하는 인자인 Index는 반드시 순서에 맞춰 0부터 N-1까지 작성하여야 한다.

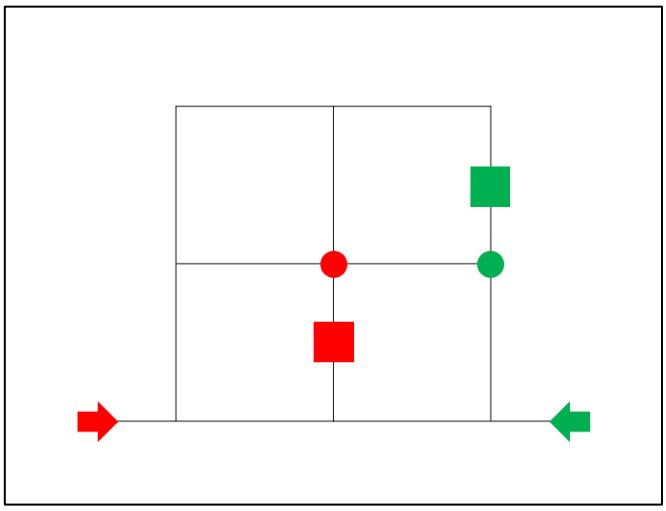
7. 예제 과정 설명 (main.py, data.py, mainProblem/testcase.py)

```
# exec problem solving
show_log = True

# solve('Route-Finding', show_log=show_log)
# print()
# solve('8-Queens', show_log=show_log, print_func=printBoard)
# print()

"""
  Main Test Problem: RGB-Express
"""
# Custom / A-4 / A-9 / B-10 / C-2 / G-8 / P-7
# rgb_testcase = 'Custom'  # UCS: 0.004, ILS: 0.004
# rgb_testcase = 'A-4'     # UCS: 0.010, ILS: 0.033
rgb_testcase = 'A-9'     # UCS: 1.452, ILS: 1.534
# rgb_testcase = 'B-10'    # UCS: 0.028, ILS: 0.102
# rgb_testcase = 'C-2'     # UCS: 18.610, ILS: 5.367
# rgb_testcase = 'G-8'     # UCS: 0.559, ILS: 1.726
# rgb_testcase = 'P-7'     # UCS: 567.446, ILS: 45.627
solve(f'RGB-Express {rgb_testcase}', show_log=show_log, print_func=printRGB)
print()
```

<RGB-Express>의 테스트 케이스들은 testcase.py에서 확인하거나 더 추가할 수 있으며, 해당 데이터는 main.py에서 테스트할 수 있다. 추가적으로 Route-Finding Problem이나 8-Queens Problem 역시 테스트할 수 있다(테스트케이스는 data.py에서 총괄한다).

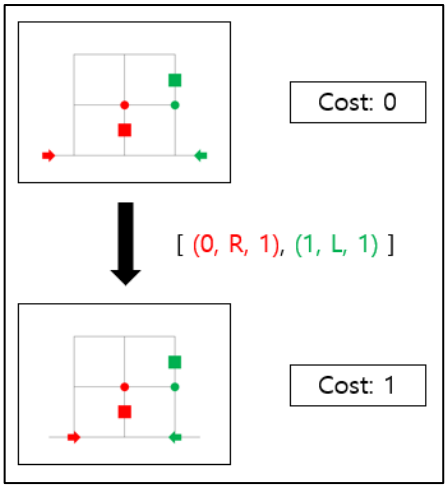


간단한 Custom Problem을 예시로 들어 Iterative Lengthening Search 알고리즘(ILS)의 작동 과정을 분석하고자 한다. 해당 문제는 상단의 그림을 바탕으로 만들어졌으며, 화살표는 트럭의 초기 상태, 동그라미는 화물, 사각형은 창고이다.

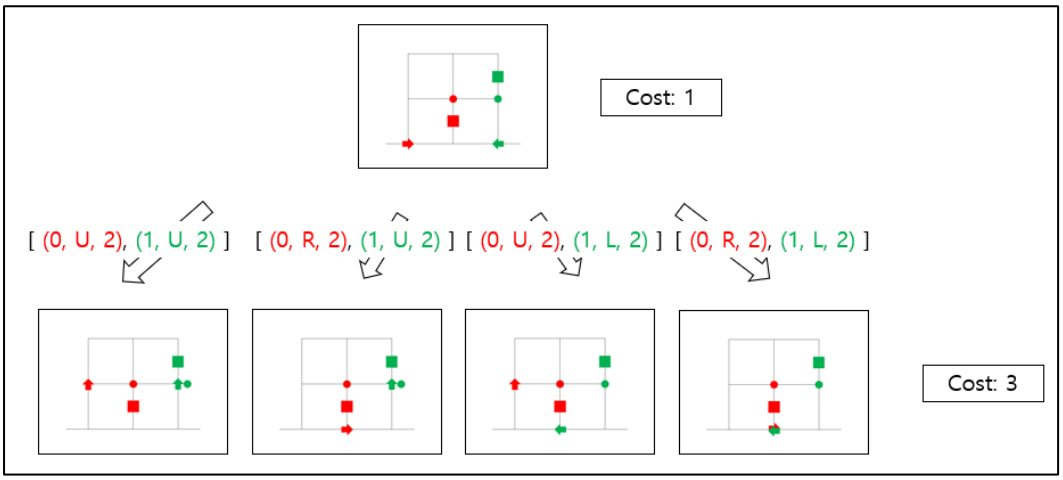
(1) Cost Limit: 0

ILS 알고리즘에서 Cost Limit의 초기값은 0이다. Initial State에서 가능한 Action은 빨간 트럭을 오른쪽으로 움직이고 초록 트럭을 왼쪽으로 움직이는 방법 밖에 없다. 해당 Action을 수행 후 Cost는 1이므로 Cutoff되어 다음 Loop로 넘어간다.

다음 Cost Limit는 Cutoff된 Cost 값 중 가장 작았던 1이다.



(2) Cost Limit: 1

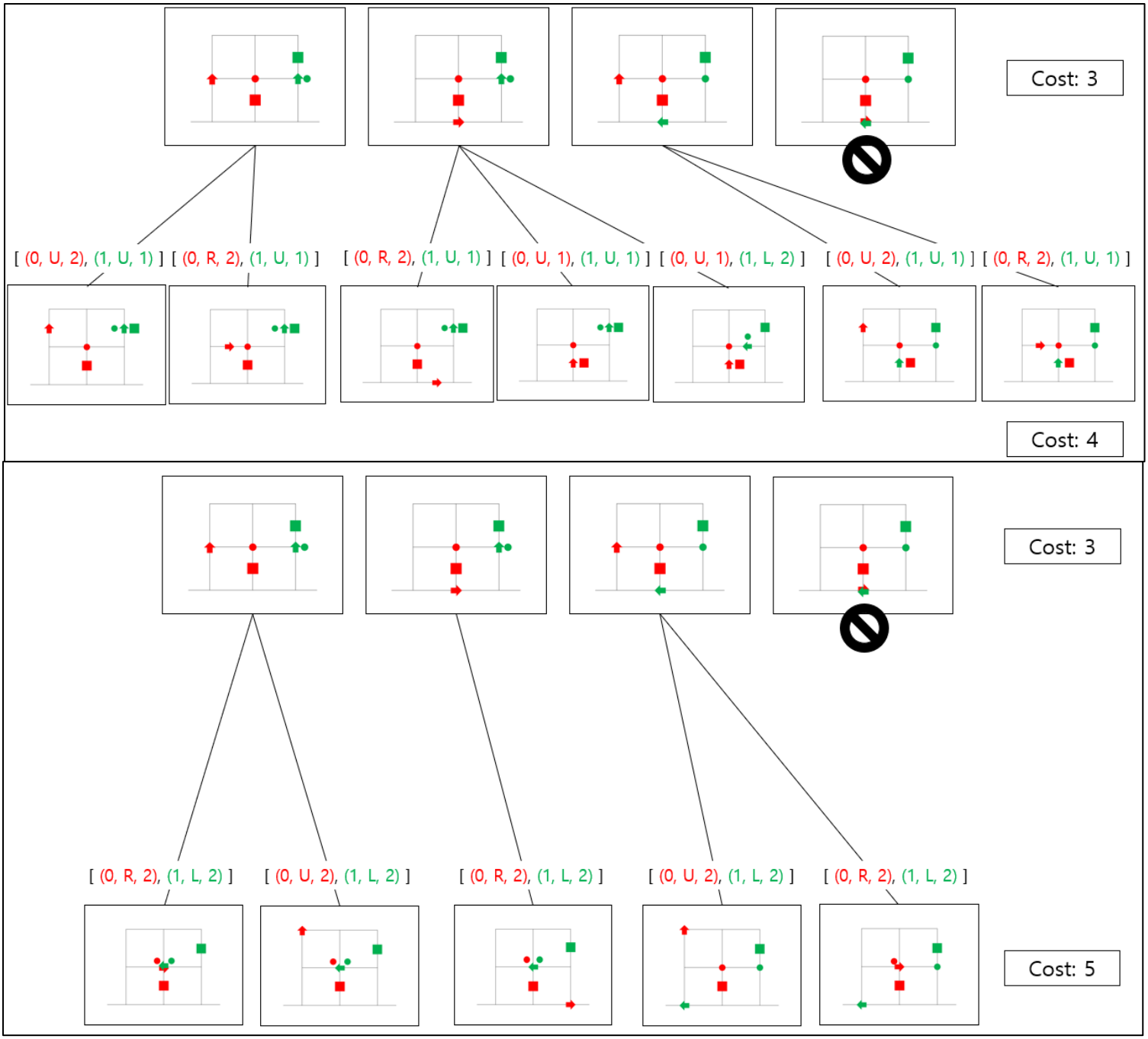


이전 Cutoff된 Cost 값 중 가장 작았던 1로 Cost Limit가 설정된다. 지면을 아끼기 위해 앞으로의 그림에서는 탐색 트리에서 이전의 Cost 부분은 계속 생략되겠지만, 실제로는 Cost 0부터 다시 시작하여 상단의 트리에 도달하는 것이므로 참조한다.

Cost 1에서는 각각의 차마다 가능한 move Dir이 2개씩이므로 총 4가지의 Action이 가능하다. Cost 3의 4번째 State의 경우 두 트럭이 충돌하여 이후 가능한 Possible Action이 존재하지 않지만, Cost Limit를 넘은 State이므로 다음 Loop에서 반영된다.

다음 Cost Limit는 Cutoff된 Cost 값 중 가장 작았던 3이다.

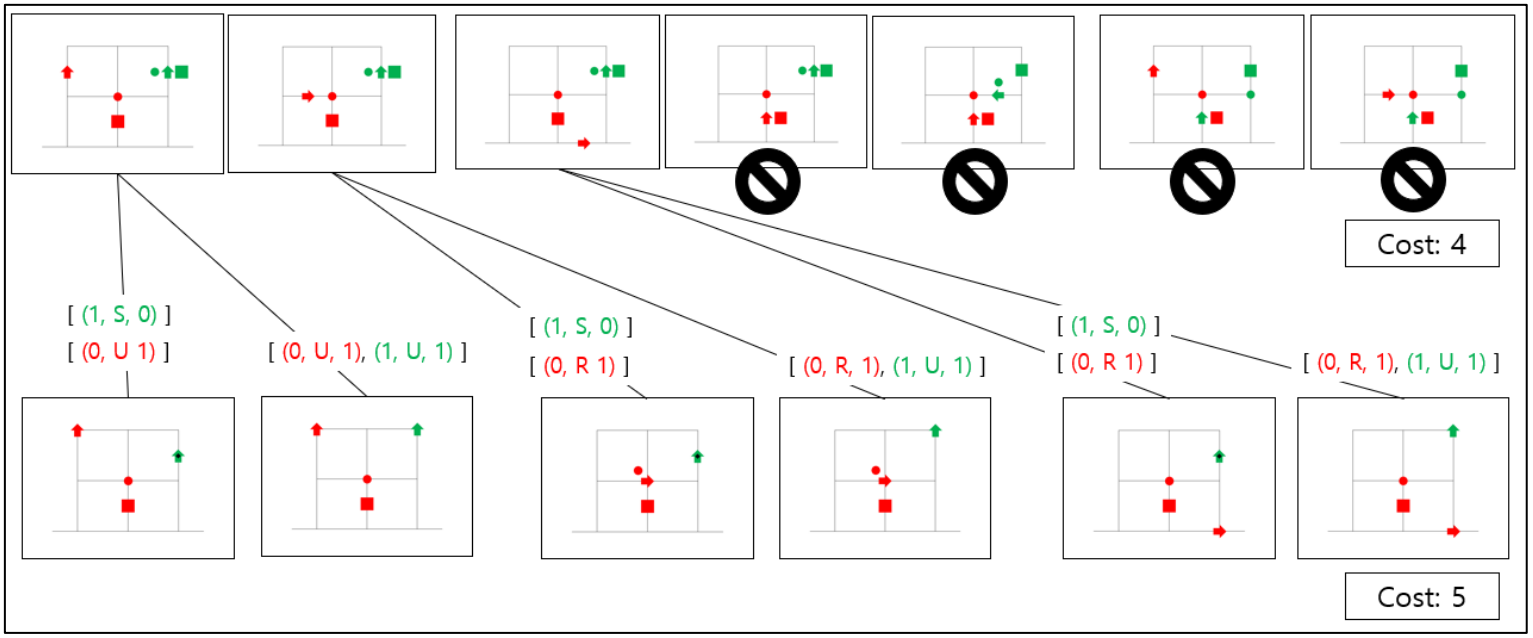
(3) Cost Limit: 3



Cost 3부터는 본격적으로 State의 개수가 많아 지기 시작한다. Cost 3에서 4번째 State는 트럭이 충돌하기 때문에(Crash Test) 가능한 Action이 존재하지 않는다. 가야 하는 길 도중에 화물이나 창고가 있을 경우에는 해당 위치에서도 RGBEvent가 발생하기 때문에 Action Cost가 1이 된다. 중간에 다른 Event가 없다면 빨간 트럭과 초록 트럭 모두 2칸을 이동할 수 있기 때문에 Action Cost가 2가 된다. 창고 Event에 도달했을 경우 Stop Action은 아직 수행할 수 없음에 유의하자. 현재 Loop에서는 Cost Limit가 3이기 때문에, Stop Action이 Cost가 0이더라도 수행하지 못 하고 Cutoff 된다.

다음 Cost Limit는 Cutoff된 Cost 값 중 가장 작았던 4이다.

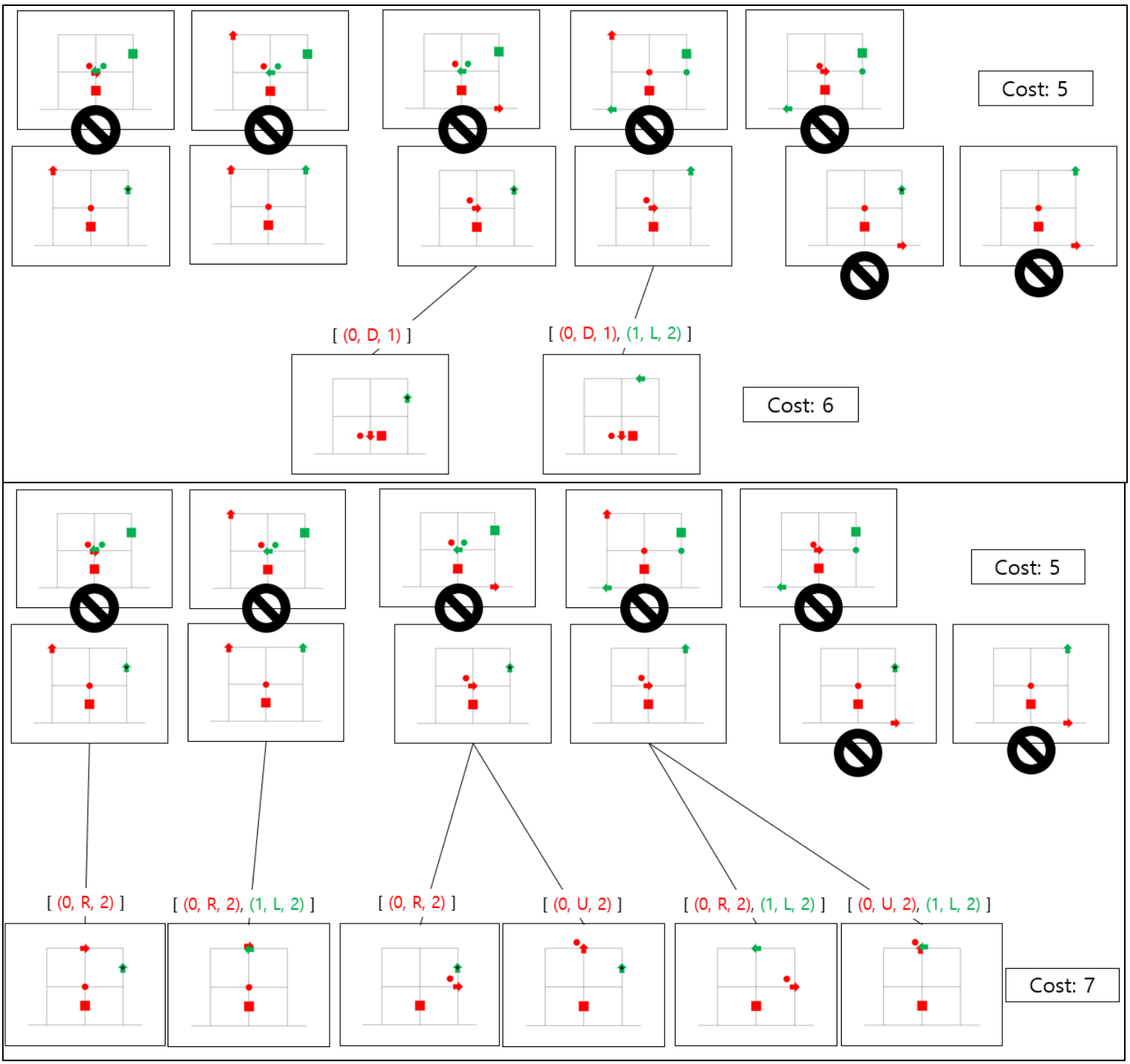
(4) Cost Limit: 4



다른 색상의 창고에 도달하거나, 화물을 내릴 수 없는 상태에서 창고에 도착할 경우에는 가능한 Action이 없다(Cargo Test)는 이유로 네 개의 State를 무시해도 된다. 만일 화물을 보유한 채로 창고에서 출발하거나 정지하는 경우(다음 Action을 수행할 경우)에는 rgbTrans 함수 내에서 화물을 창고에 내리는 과정을 수행한다. 모든 State에서 빨간 트럭의 remain이 1이기 때문에 이번 Loop에서 최대 Action Cost는 1이다.

다음 Cost Limit는 Cutoff된 Cost 값 중 가장 작았던 5이다.

(5) Cost Limit: 5

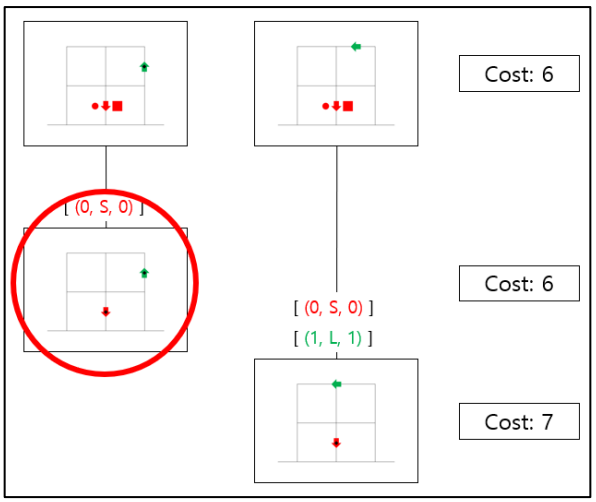


두 트럭이 충돌하거나(Crash Test), 다른 색의 아이템에 접근하거나(Item Test), 나갈 수 있는 Dir이 없는 교차로에 진입하는(Intersection Test) 등의 이유로, Cost 5의 많은 State들에서 Possible Action이 존재하지 않는다. 초록 트럭이 정지한 State에서는 Action이 빨간색 트럭의 Move만 포함하는 것을 확인할 수 있다.

다음 Cost Limit는 Cutoff된 Cost 값 중 가장 작았던 6이다.

(6) Cost Limit: 6

Cost 6의 첫 번째 State에서 빨간색 트럭이 Stop Action을 수행하면 모든 트럭이 멈추고 모든 창고가 채워졌으므로 Goal State가 된다. 따라서 탐색을 종료하고 Best Cost인 6과 Solution Path인




```
[
    [ (0, R, 1), (1, L, 1) ],
    [ (0, U, 2), (1, U, 2) ],
    [ (0, R, 2), (1, U, 1) ],
    [ (1, S, 0) ],
    [ (0, R, 1) ],
    [ (0, D, 1) ],
    [ (0, S, 0) ]
]
```

```
    Find Cost: 0
    Find Cost: 1
    Find Cost: 3
    Find Cost: 4
    Find Cost: 5
    Find Cost: 6
< Optimal Cost 6 >
Iterative Lengthening Search Result: 0.003989219665527344
[Truck 0] →↑.→.↓*
[Truck 1] ←↑.↑*
```

을 반환한다. 실제 알고리즘의 수행 결과 역시 같은 결과를 도출하는 것을 확인할 수 있다.

이 다음 페이지에 부록으로, 실제 <RGB-Express> 문제를 형식화하여 Solution Path를 찾아내는 예시를 볼 수 있다.

8. 참조

RGB-Express 문제 이미지 자료: https://www.rgbexpresswalkthrough.com/

9. 부록

A-9


```
'A-9': {
    # road tokens: '-', '-', '╵', '╷', '─', '│', '┌', '┐', '┘', '└', '├',
    'Roads': "╷     ╷\n"  # 0
             "├─┬─┬─┤\n"  # 1
             "│ │ │ │\n"  # 2
             "├─┼─┼─┤\n"  # 3
             "│ ├─┤ │\n"  # 4
             "├─┤ ├─┤\n"  # 5
             "│ │ │ │\n"  # 6
             "├─┴─┴─┤\n"  # 7
             "╵     ╵",   # 8
    'Trucks': [
        Truck(0, Color.Red, Dir.Up, (8, 0)),
        Truck(1, Color.Yellow, Dir.Up, (8, 6))
    ],
    'Items': {
        Color.Red: [(3, 2), (5, 2)],
        Color.Yellow: [(5, 4)],
    },
    'Cargos': {
        Color.Red: [(0, 0), (1, 3)],
        Color.Yellow: [(0, 6)],
    }
'A-9'
```

```
Run: main
< Optimal Cost 20 >
Uniform Search Result: 1.4521172046661377
[Truck 0] ↑→.↑.↑→.↑↑.←↓.←.↑.↑*
[Truck 1] ↑←.↑.→.↑.↑.↑*

< Optimal Cost 20 >
Iterative Lengthening Search Result: 1.5338971614837646
[Truck 0] ↑↑.→.↑→.↑↑.←↓.←.↑.↑*
[Truck 1] ↑←.↑.→.↑.↑.↑*

############### End ###############
```

G-8


```
'G-8': {
    # road tokens: '-', '-', '╵', '╷', '─', '│', '┌', '┐', '┘', '└', '├',
    'Roads': "┬┬┬┬┬┬┬\n"  # 0
             "│ │ │ │ │\n"  # 1
             "├┼┼┼┼┼┼┤\n"  # 2
             "│ │ │ │ │\n"  # 3
             "├┼┼┼┼┼┼┤\n"  # 4
             "│ │ │ │ │\n"  # 5
             "├┼┼┼┼┼┼┤\n"  # 6
             "│ │ │ │ │\n"  # 7
             "╵ ╵ ╵ ╵ ╵",  # 8
    'Trucks': [
        Truck(0, Color.Red, Dir.Up, (8, 1)),
        Truck(1, Color.Blue, Dir.Up, (8, 5)),
        Truck(2, Color.Green, Dir.Up, (8, 9)),
    ],
    'Items': {
        Color.Red: [(4, 1)],
        Color.Blue: [(2, 4), (4, 9)],
        Color.Green: [(3, 3)],
    },
    'Cargos': {
        Color.Red: [(0, 6)],
'G-8'
```

```
Run: main
C:\Users\LG\PycharmProjects\AIMidterm\venv\Scripts\python.exe C:/Users/LG/PycharmProjects/AIMidterm/main.py
##### RGB-Express G-8 Problem #####

< Optimal Cost 25 >
Uniform Search Result: 0.5585088729858398
[Truck 0] ↑.↑.→.→.↑.↑.→*
[Truck 1] ↑.↑.→.→.↑.↑.←↓.←.←←.↑.→*
[Truck 2] ↑.←.←.←.↑.↑↑.→*

< Optimal Cost 25 >
Iterative Lengthening Search Result: 1.7261378765106201
[Truck 0] ↑.↑.→.→.↑.↑.→*
[Truck 1] ↑.↑.→.→.↑.↑.←↓.←.←←.↑.→*
[Truck 2] ↑.←.←.←.↑.↑↑.→*

############### End ###############
```