Domain파일

predicate과 action이 정의되어 실세계의 현상을 추상화 한다.

```
(define (domain dock-worker-robot-pos)
 (:requirements :strips :typing )
 (:types
  location        ; there are several connected locations in the harbor
  pile                ; is attached to a location
                          ; it holds a pallet and a stack of containers
  robot             ; holds at most 1 container, only 1 robot per location
  crane             ; belongs to a location to pickup containers
  container)

 (:constants
  pallet - container)
```

먼저 첫번째 줄에 해당 domain 명을 정의한다. 여기서는 dock-worker-robot-pos가 도메인 명이 된다. 이를 통해서 problem파일과 연결할 수 있게 된다.

Requirements는 파이썬의 import나 C의 include와 비슷한 것이라고 한다. Strips는 스탠포드 문법을 사용할 수 있게 하고, typing은 클래스를 정하는 것과 비슷하다고 설명되어 있다. 그래서 다음에 types가 나온다. Type은 클래스들이라고 생각할 수 있다. 이것을 사용하는 이유를 생각해보면 gripper 예제를 보면 predicate중에 (room ?r)이 있었다. 이런 것을 predicate로 작성하지 않고 타입으로 바꾸어 ?r – room 이런 식으로 strips 형식으로 바꿔서 작성하는 것으로 보인다. 좀 더 가독성이 좋아지는 것 같다.

Constants는 문제에서 모든 State에 존재하는 object라고 생각하면 된다. 일반적으로 잘 쓰진 않는다고 한다.

```
(:predicates
    (adjacent ?l1   ?l2 - location)          ; location ?l1 is adjacent ot ?l2
    (attached ?p - pile ?l - location)   ; pile ?p attached to location ?l
    (belong ?k - crane ?l - location)     ; crane ?k belongs to location ?l

    (atl ?r - robot ?l - location)          ; robot ?r is at location ?l
    (free ?l - location)                    ; there is a robot at location ?l
    (loaded ?r - robot ?c - container ) ; robot ?r is loaded with container ?c
    (unloaded ?r - robot)                       ; robot ?r is empty

    (holding ?k - crane ?c - container) ; crane ?k is holding a container ?c
    (empty ?k - crane)                                  ; crane ?k is empty

    (in ?c - container ?p - pile)            ; container ?c is within pile ?p
    (top ?c - container ?p - pile)         ;   container ?c is on top of pile ?p
    (on ?k1 - container ?k2 - container); container ?k1 is on container ?k2
    )
```

다음으로 predicate를 살펴보자. 어떤 상태를 표현한다고 볼 수 있다.

- Adjacent: 두 location이 인접해있다.
- Attached: pile(컨테이너를 쌓아놓은 것)이 location에 있다.
- Belong: 크레인이 location에 있다.
- Atl: 로봇이 location에 있다.
- Free: 로봇이 location에 없다 (교재에는 occupied로 되어 있어서 주석이 "로봇이 location에 있다"로 적혀있다.)
- Loaded: 로봇이 컨테이너를 싣고 있다.
- Unloaded: 로봇이 아무것도 싣지 않고 있다.
- Holding: 크레인이 컨테이너를 잡고 있다.
- Empty: 크레인이 놀고 있다.
- In: 컨테이너가 pile에 있다.
- Top: 컨테이너가 pile의 꼭대기에 있다.
- On: 컨테이너가 다른 컨테이너 위에 있다.

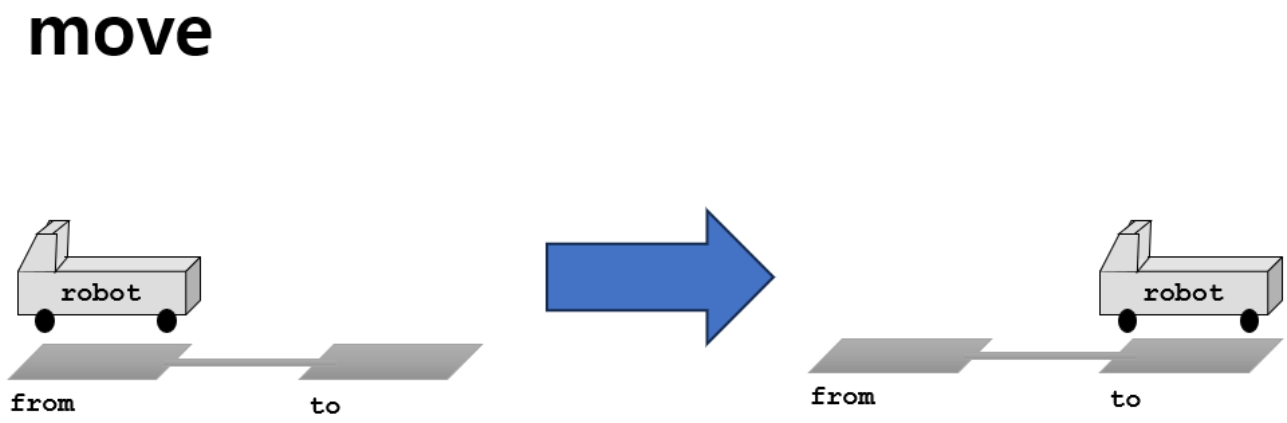
이런 predicate를 통해 참인 것을 표현하고 그들을 모아서 State를 표현한다.

```
;; moves a robot between two adjacent locations
 (:action move
       :parameters (?r - robot ?from ?to - location)
       :precondition (and (adjacent ?from ?to)
                          (atl ?r ?from) (free ?to))
       :effect (and (atl ?r ?to) (free ?from)
                                  (not (free ?to)) (not (atl ?r ?from)) ))

;; loads an empty robot with a container held by a nearby crane
 (:action load
       :parameters (?k - crane ?l - location ?c - container ?r - robot)
       :precondition (and (atl ?r ?l) (belong ?k ?l)
                          (holding ?k ?c) (unloaded ?r))
       :effect (and   (loaded ?r ?c) (not (unloaded ?r))
                      (empty ?k) (not (holding ?k ?c))))


;; unloads a robot holding a container with a nearby crane
 (:action unload
       :parameters (?k - crane ?l - location ?c - container ?r - robot)
       :precondition (and (belong ?k ?l) (atl ?r ?l)
                          (loaded ?r ?c) (empty ?k))
       :effect (and (unloaded ?r) (holding ?k ?c)
                    (not (loaded ?r ?c))(not (empty ?k))))

;; takes a container from a pile with a crane
 (:action take
```

```
    :parameters (?k - crane ?l - location ?c ?else - container ?p - pile)
    :precondition (and (belong ?k ?l)(attached ?p ?l)
                       (empty ?k) (in ?c ?p)
                       (top ?c ?p) (on ?c ?else))
    :effect (and (holding ?k ?c) (top ?else ?p)
                 (not (in ?c ?p)) (not (top ?c ?p))
                 (not (on ?c ?else)) (not (empty ?k))))

;; puts a container held by a crane on a nearby pile
 (:action put
    :parameters (?k - crane ?l - location ?c ?else - container ?p - pile)
    :precondition (and (belong ?k ?l) (attached ?p ?l)
                       (holding ?k ?c) (top ?else ?p))
    :effect (and (in ?c ?p) (top ?c ?p) (on ?c ?else)
                 (not (top ?else ?p)) (not (holding ?k ?c))
                 (empty ?k))))
```

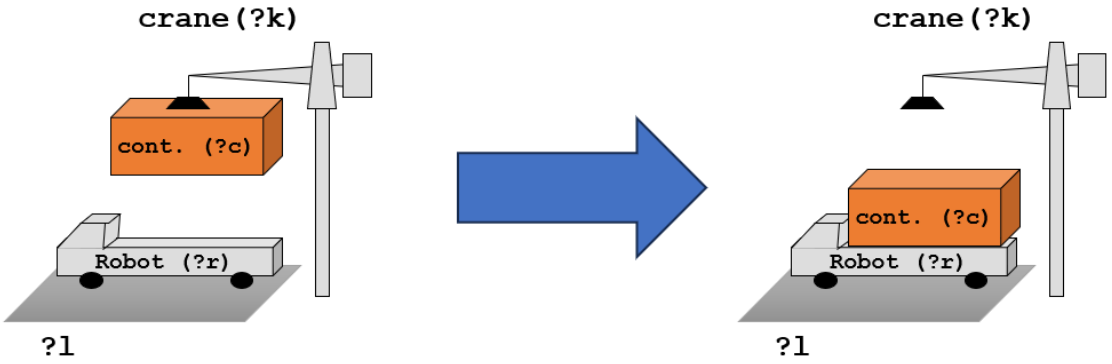
Action

- Move: 로봇이 from에서 to로 이동한다.
  - 파라미터: 로봇(?r), 출발지(?from), 도착지(?to)
  - Precondition: from과 to는 인접해야하며, 로봇은 from에 위치하고 to는 비어있어야 한다.
  - Effect: 로봇이 to에 있게 되고, from은 비게 된다. 그리고 to는 더 이상 비어있지 않고 로봇은 from에 위치하지 않게 된다.



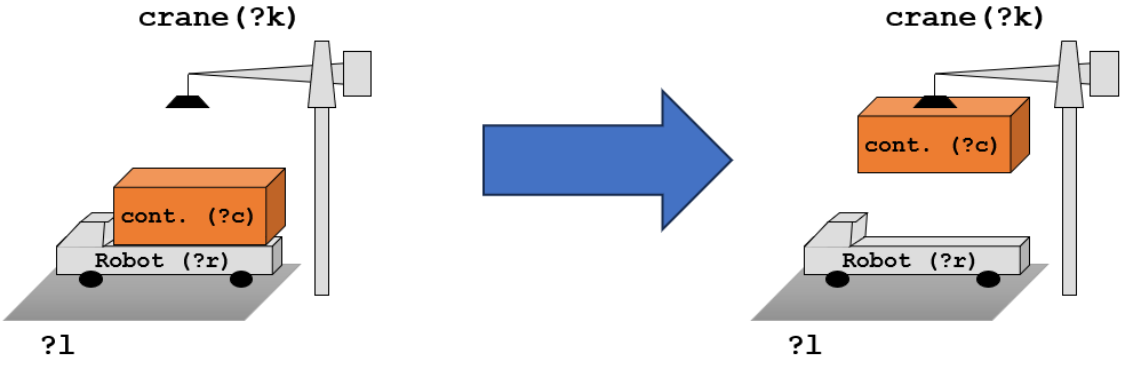
- Load: location에서 크레인이 컨테이너를 로봇에 싣는다.
  - 파라미터: 크레인(?k), 위치(?l), 컨테이너(?c), 로봇(?r)
  - Precondition: 로봇이 위치에 있고, 크레인도 위치에 있어야 하며 크레인이 컨테이너를 잡고 있어야 하며 로봇은 아무것도 싣고 있지 않다.
  - Effect: 로봇이 컨테이너를 싣고 있고 더 이상 로봇이 안 싣고 있지 않다. 그리고 크레인은 이제 비어있으며 컨테이너를 잡고 있지 않게 되었다.

Load



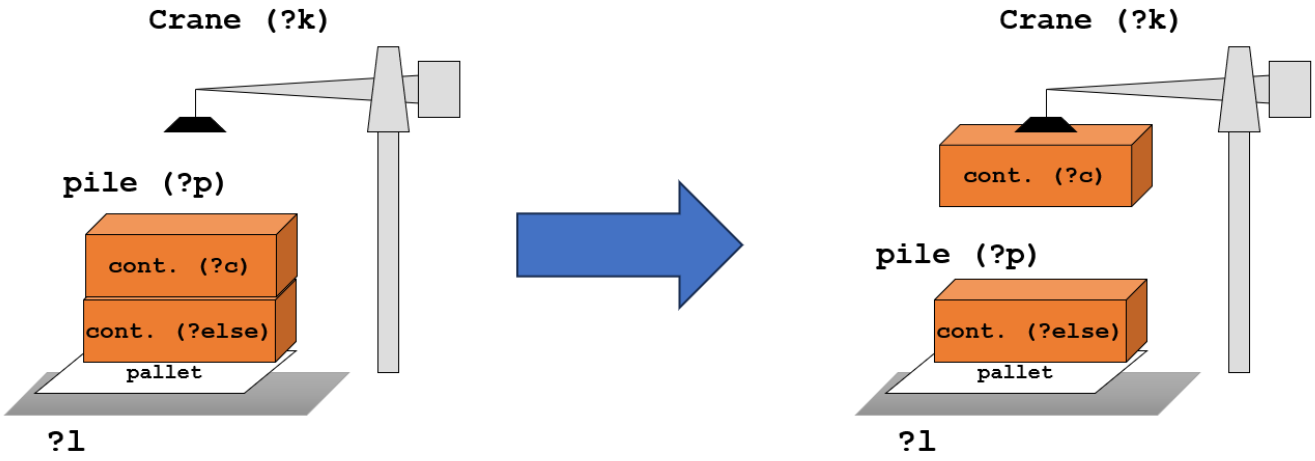
- Unload: location에서 크레인이 로봇에서 컨테이너를 뺀다.
  - 파라미터: 크레인(?k), 위치(?l), 컨테이너(?c), 로봇(?r)
  - Precondition: 로봇이 위치에 있고, 크레인도 위치에 있어야 하며 로봇이 컨테이너를 싣고 있어야 하며 크레인은 아무것도 잡고 있지 않다.
  - Effect: 이제 로봇은 아무것도 싣지 않으며 크레인은 컨테이너를 잡고 있다. 그리고 로봇은 더 이상 컨테이너를 싣고 있지 않으며 크레인도 더 이상 비어 있지 않다.

unload



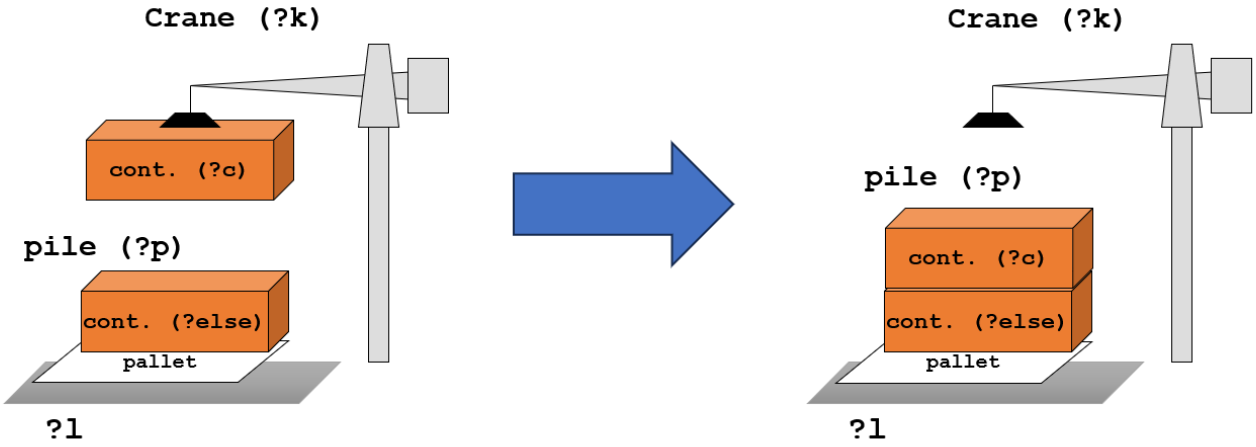
- Take: location에서 크레인이 else 컨테이너 위에 있는 c컨테이너를 가져온다.
  - 파라미터: 크레인(?k), 위치(?l), 컨테이너(?c, ?else), 컨테이너 탑(?p)
  - Precondition: 크레인이 위치에 있어야 하며 pile이 위치에 있고 크레인은 아무것도 잡고 있지 않다. 그리고 c컨테이너는 pile에 속하며 가장 위에 있다. 그리고 c는 else 위에 있다.
  - Effect: 이제 크레인은 c 컨테이너를 잡고 있고 pile의 꼭대기는 else컨테이너가 된다. 그리고 c는 더 이상 pile에 속하지 않고 꼭대기도 아니고 else위에 있지도 않다. 그리고 크레인은 더 이상 비어 있지 않다.

Take



- Put: location에서 크레인이 else위에 c 컨테이너를 쌓는다.
  - 파라미터: 크레인(?k), 위치(?l), 컨테이너(?c, ?else), 컨테이너 탑(?p)
  - Precondition: 크레인이 위치에 있어야 하며 pile이 위치에 있고 크레인은 c컨테이너를 잡고 있고 pile의 꼭대기는 else 컨테이너이다.
  - Effect: 이제 c는 pile에 속하고 꼭대기에 있으며 else 위에 있다. 그리고 else는 더 이상 pile의 top이 아니고 크레인은 c를 잡고 있지 않고 비어있다.

Put



- Problem 파일

Problem 파일은 도메인 파일에서 정의된 predicate를 이용해 실제 상황을 표현한다. Objects는 실제 상황에 주어진 인스턴스들이며 predicate의 변수에 objects를 집어넣어 ground시킨다. Ground된 predicate들을 모아서 state를 표현할 수 있다. 이런 방식으로 초기 state를 init으로 표현하고, 목표 state를 goal로 표현한다.

(define (problem dwrpb1)

```
(:domain dock-worker-robot-pos)

(:objects
 r1 - robot
 l1 l2 - location
 k1 k2 - crane
 p1 q1 p2 q2 - pile
 ca cb cc cd ce cf - container)

(:init
 (adjacent l1 l2)
 (adjacent l2 l1)

 (attached p1 l1)
 (attached q1 l1)
 (attached p2 l2)
 (attached q2 l2)

 (belong k1 l1)
 (belong k2 l2)

 (in ca p1)
 (in cb p1)
 (in cc p1)

 (in cd q1)
 (in ce q1)
 (in cf q1)

 (on ca pallet)
 (on cb ca)
 (on cc cb)

 (on cd pallet)
 (on ce cd)
 (on cf ce)

 (top cc p1)
 (top cf q1)
 (top pallet p2)
 (top pallet q2)

 (atl r1 l1)
 (unloaded r1)
```

```
    (free l2)

    (empty k1)
    (empty k2))

;; the task is to move all containers to locations l2
;; ca and cc in pile p2, the rest in q2
  (:goal
    (and (in ca p2)
                 (in cb q2)
                 (in cc p2)
                 (in cd q2)
                 (in ce q2)
                 (in cf q2))))
```

앞에서부터 살펴보겠다.

```
(define (problem dwrpb1)
  (:domain dock-worker-robot-pos)

  (:objects
    r1 - robot
    l1 l2 - location
    k1 k2 - crane
    p1 q1 p2 q2 - pile
    ca cb cc cd ce cf - container)
```

먼저 problem의 이름을 지정한다. 그렇게 중요한 부분은 아니다. 그 다음 domain을 선언한다. 해당 문제가 어느 도메인에 있는지 말하는 것이며 domain파일에 언급된 이름과 동일해야 planner가 정상적으로 동작하는 것으로 보인다. 그 다음으로 objects를 선언한다. 문제가 갖고 있는 인스턴스들이다. 한 줄에 객체 명 – 객체 타입의 형태로 선언하며 객체 명은 여러 개가 나올 수 있다. 그러면 같은 줄에 있으면 같은 타입으로 인식한다. 이를 해석하면 로봇 1기, 위치 2곳, 크레인 2기, 컨테이너를 쌓을 수 있는 장소 4곳, 컨테이너 6개가 존재한다.

```
(:init
    (adjacent l1 l2)
    (adjacent l2 l1)

    (attached p1 l1)
    (attached q1 l1)
    (attached p2 l2)
    (attached q2 l2)

    (belong k1 l1)
    (belong k2 l2)
```

(in ca p1)
(in cb p1)
(in cc p1)

(in cd q1)
(in ce q1)
(in cf q1)

(on ca pallet)
(on cb ca)
(on cc cb)

(on cd pallet)
(on ce cd)
(on cf ce)

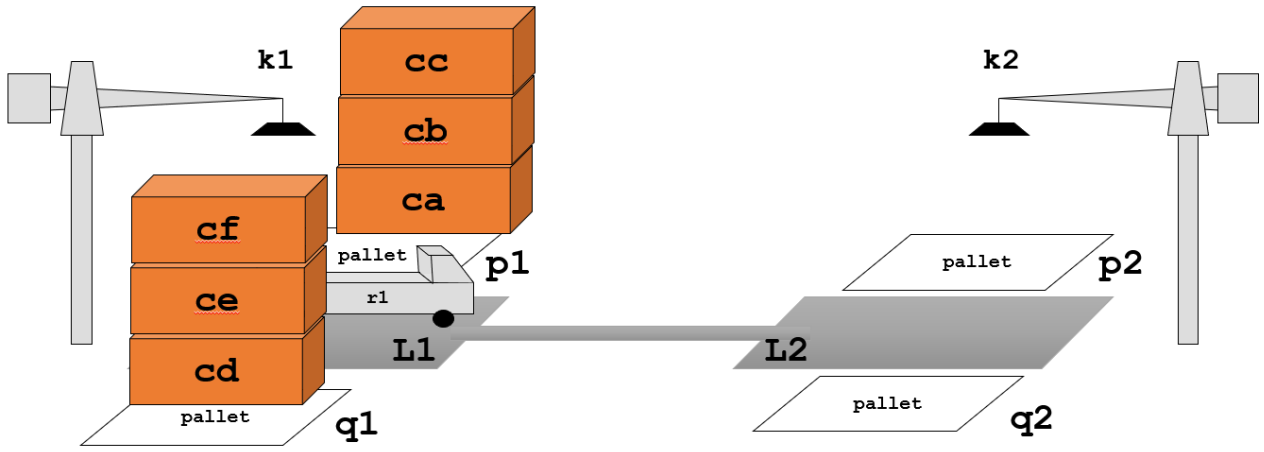
(top cc p1)
(top cf q1)
(top pallet p2)
(top pallet q2)

(atl r1 l1)
(unloaded r1)
(free l2)

(empty k1)
(empty k2))

Init을 통해 초기 state를 표현할 수 있다. 순서대로 해석하면 l1과 l2는 인접해있다. 컨테이너 더미 p1, q1은 l1에 있고 p2, q2는 l2에 있다. 그리고 크레인k1, k2가 l1, l2에 각각 위치한다. 컨테이너 ca, cb, cc는 p1에 속하며 cd, ce, cf는 q1에 속한다. P1은 pallet-ca-cb-cc 순으로 쌓여 있고 p2는 pallet-cd-ce-cf 순으로 쌓여 있다. 따라서 p1의 top은 cc, q1의 top은 cf, p2와 q2의 top은 pallet이다. 로봇 r1은 l1에 있으며 싣고 있는 것은 없다. L2는 비어있다. 그리고 크레인도 현재 사용중이 아니다.
Init은 자세하게 모든 상황을 정의하는 것을 볼 수 있다.

Init state



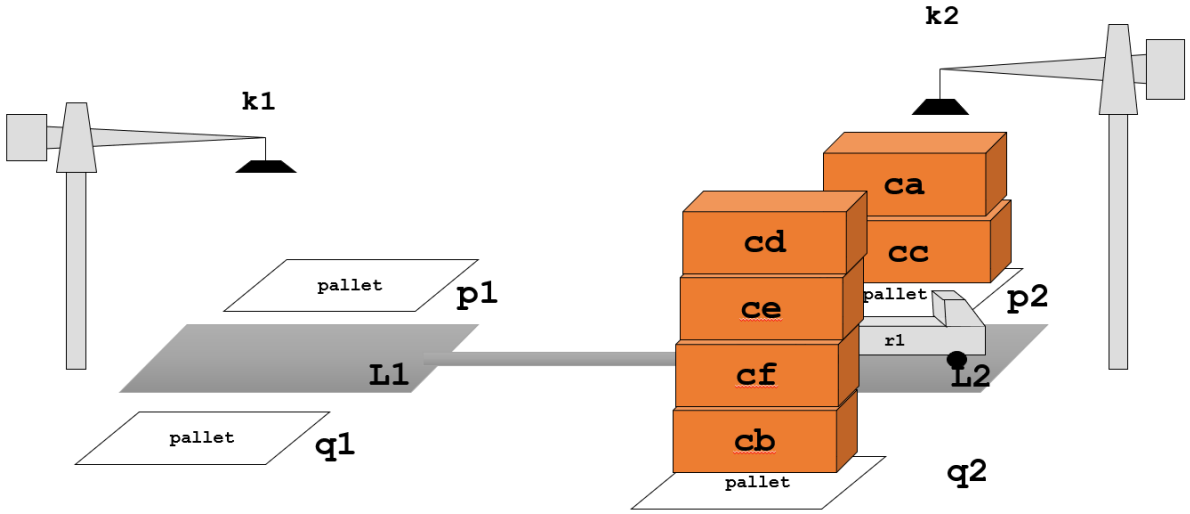
(:goal

(and (in ca p2) (in cc p2)

(in cb q2) (in cd q2) (in ce q2) (in cf q2))))

마지막으로 goal을 살펴보면 p2에 ca, cc를 쌓고 q2에 cb, cd, ce, cf가 쌓여 있으면 된다. 순서는 언급되지 않았으므로 속하기만 하면 된다. 아래 그림은 그 중 하나이며 쌓은 순서 상관없이 각 컨테이너가 원하는 더미에 쌓여 있기만 하면 우리는 목표 state에 도달한 것이다.

Goal state



Domain과 problem 파일을 작성했으니 planner에 넣고 결과를 살펴보자.

FF planner를 사용했다. FF planner를 사용하는 방법은 다음과 같다.

ff -o <domain파일> -f <problem파일>

```
kim@DESKTOP-V1H4KFU:~$ ff -o ~/domain/domain-02.pddl -f ~/problem/problem-02.pddl
```

```
ff: parsing domain file
domain 'DOCK-WORKER-ROBOT-POS' defined
 ... done.
ff: parsing problem file
problem 'DWRPB1' defined
 ... done.
```

FF planner가 domain 파일과 problem 파일을 성공적으로 파싱하면 도메인 명과 problem 명이 정의되었다고 출력된다.

```
Cueing down from goal distance:   25 into depth [1]
                                  24             [1]
                                  23             [1][2]
                                  22             [1]
                                  21             [1]
                                  20             [1][2]
                                  19             [1][2]
                                  18             [1]
                                  17             [1]
                                  16             [1][2]
                                  15             [1][2]
                                  14             [1]
                                  13             [1]
                                  12             [1][2]
                                  11             [1][2]
                                  10             [1]
                                   9             [1]
                                   8             [1][2]
                                   7             [1]
                                   6             [1][2]
                                   5             [1]
                                   4             [1][2]
                                   3             [1]
                                   2             [1]
                                   1             [1]
                                   0
```

위 사진은 planner의 검색 과정이다. GPT에 물어본 바로는 목표 거리와 거리를 줄이는데 필요한 단계 수라고 한다. 이 문제는 처음에 25의 거리를 갖고 있으며 25에서 24로 줄이는데 1단계가 필요하다. [1][2]로 되어 있는 경우는 2단계가 필요하다는 것이라고 한다.

플래너가 내놓은 답을 따라가보니 MOVE가 끼어 있는 경우들이 2단계가 필요한 것으로 확인했다. MOVE만으로는 goal distance를 줄이지 못하는 모양이다.

```
ff: found legal plan as follows

step    0: TAKE K1 L1 CC CB P1
        1: LOAD K1 L1 CC R1
        2: MOVE R1 L1 L2
        3: UNLOAD K2 L2 CC R1
        4: PUT K2 L2 CC PALLET P2
        5: TAKE K1 L1 CB CA P1
        6: MOVE R1 L2 L1
        7: LOAD K1 L1 CB R1
        8: MOVE R1 L1 L2
        9: UNLOAD K2 L2 CB R1
       10: PUT K2 L2 CB PALLET Q2
       11: TAKE K1 L1 CA PALLET P1
       12: MOVE R1 L2 L1
       13: LOAD K1 L1 CA R1
       14: MOVE R1 L1 L2
       15: UNLOAD K2 L2 CA R1
       16: PUT K2 L2 CA CC P2
       17: TAKE K1 L1 CF CE Q1
       18: MOVE R1 L2 L1
       19: LOAD K1 L1 CF R1
       20: MOVE R1 L1 L2
       21: UNLOAD K2 L2 CF R1
       22: PUT K2 L2 CF CB Q2
       23: TAKE K1 L1 CE CD Q1
       24: MOVE R1 L2 L1
       25: LOAD K1 L1 CE R1
       26: TAKE K1 L1 CD PALLET Q1
       27: MOVE R1 L1 L2
       28: UNLOAD K2 L2 CE R1
       29: PUT K2 L2 CE CF Q2
       30: MOVE R1 L2 L1
       31: LOAD K1 L1 CD R1
       32: MOVE R1 L1 L2
       33: UNLOAD K2 L2 CD R1
       34: PUT K2 L2 CD CE Q2
```

그 다음으로 플래너가 goal state까지의 경로를 찾은 경우 그 답을 보여준다. 순서대로 해석해보자

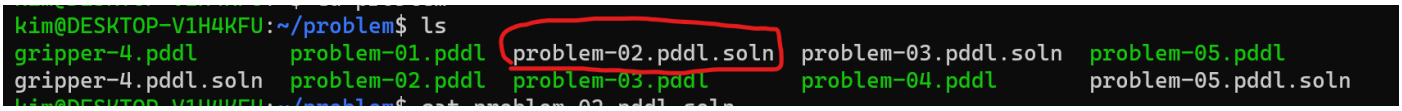
- 0~4: CC를 p2로 옮김
- 5~10: CB를 q2로 옮김
- 11~16: CA를 p2로 옮김
- 17~22: CF를 q2로 옮김
- 23~29: CE를 q2로 옮김
- 30~34: CD를 q2로 옮김

위와 같은 순서로 진행된다. 어느정도 순서가 정해져서 반복되는데 26에서는 CE를 로봇에 load한 다음 move전에 take를 해서 크레인이 먼저 컨테이너를 잡고 있는 경우도 생겼다.

```
time spent:    0.00 seconds instantiating 422 easy, 0 hard action templates
               0.00 seconds reachability analysis, yielding 119 facts and 362 actions
               0.00 seconds creating final representation with 119 relevant facts
               0.00 seconds building connectivity graph
               0.00 seconds searching, evaluating 73 states, to a max depth of 2
               0.00 seconds total time
```

마지막으로는 시간 측정이 이뤄진다.

그리고 해답은 problem파일이 위치한 곳에 생기게 된다. Planner 종료 후 problem파일이 있는 곳을 보면 soln파일이 생긴 것을 확인할 수 있다.



내용을 확인하면 위에 출력되었던 해답이 저장되어 있는 것을 확인할 수 있다.

```
kim@DESKTOP-V1H4KFU:~/problem$ cat problem-02.pddl.soln
Time 0
(TAKE K1 L1 CC CB P1)
(LOAD K1 L1 CC R1)
(MOVE R1 L1 L2)
(UNLOAD K2 L2 CC R1)
(PUT K2 L2 CC PALLET P2)
(TAKE K1 L1 CB CA P1)
(MOVE R1 L2 L1)
(LOAD K1 L1 CB R1)
(MOVE R1 L1 L2)
(UNLOAD K2 L2 CB R1)
(PUT K2 L2 CB PALLET Q2)
(TAKE K1 L1 CA PALLET P1)
(MOVE R1 L2 L1)
(LOAD K1 L1 CA R1)
(MOVE R1 L1 L2)
(UNLOAD K2 L2 CA R1)
(PUT K2 L2 CA CC P2)
(TAKE K1 L1 CF CE Q1)
(MOVE R1 L2 L1)
(LOAD K1 L1 CF R1)
(MOVE R1 L1 L2)
(UNLOAD K2 L2 CF R1)
(PUT K2 L2 CF CB Q2)
(TAKE K1 L1 CE CD Q1)
(MOVE R1 L2 L1)
(LOAD K1 L1 CE R1)
(TAKE K1 L1 CD PALLET Q1)
(MOVE R1 L1 L2)
(UNLOAD K2 L2 CE R1)
(PUT K2 L2 CE CF Q2)
(MOVE R1 L2 L1)
(LOAD K1 L1 CD R1)
(MOVE R1 L1 L2)
(UNLOAD K2 L2 CD R1)
(PUT K2 L2 CD CE Q2)
```