플래닝을 이용한 워터솔트 퍼즐 PLAN 도출

게임소프트웨어공학과

B977082 백정열

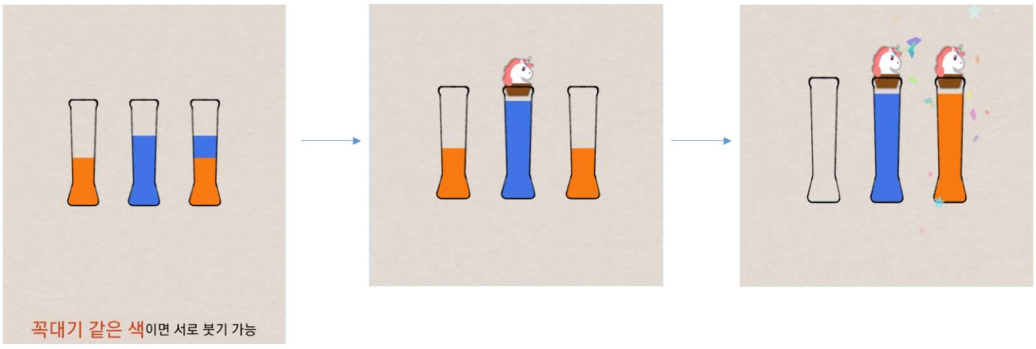
TITLE

플래닝으로 이용한 워터솔트 퍼즐 PLAN 도출

watersort puzzle plan derive using planning

I.서론

워터솔트 퍼즐은 최근 웹, 유튜브등에 많이 광고되고 있는 모바일 퍼즐게임이다. 먼저, 아래는 워터솔트 퍼즐의 간단한 설명이다.



워터 솔트퍼즐은 간단하게, 같은병에 같은색 물만 가득 차있게 만들면 되는 게임이다. 병의 숫자가 늘어날수록 선택지는 늘어나고, 잘못된 선택을 이어가다 보면 더 이상 물을 움직일 수 없는 상황이 나오게 된다.

'워터솔트 퍼즐의 시도한 횟수와 걸린시간 등을 스코어로 환산하여 대전게임 형식으로 발전시킨다면, 플래닝으로 plan을 도출하였을때 최고점수를 확인 할 수 있지 않을까?' 라는 생각에서 발전되어 이 프로젝트를 진행하게 되었다.

이 프로젝트는 워터솔트 퍼즐 문제가 주어졌을때 PDDL domain과 문제가 적용된 problem에서 goal을 달성할 plan을 제시한다. 플래닝으로 작성되었기 때문에 goal을 다양하게 설정할 수 있다. 워터솔트 퍼즐을 예로 들면, 각 병의 물을 같은색으로만 설정하지않고, 각 병에 물의 색이 2가지만 들어가있게 goal을 설정하여 plan을 도출 할 수 있다. 이를 응용하면 스테이지를 설계할때도 사용가능하다. 개발자가 스테이지를 설계하였을때, 스테이지를 클리어할 plan이 없다면 잘못된 스테이지 일것이며, 스테이지를 클리어 할 수 있는 plan이 너무 많다면 너무 쉬운 스테이지가 될 것이다. 해당 프로젝트를 참고하면 스테이지 설계에도 큰 도움이 될 것이라고 생각한다.

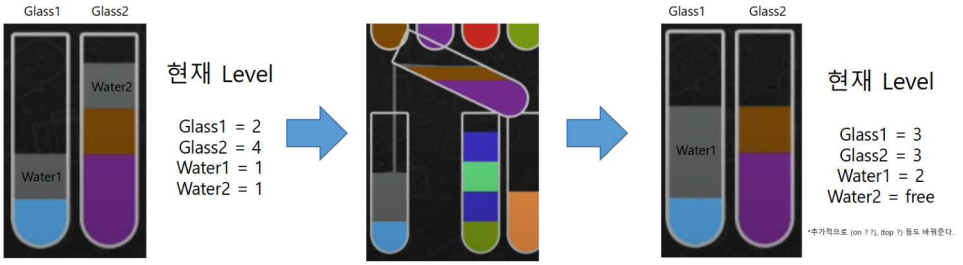
II.본론

워터 솔트 퍼즐을 플래닝으로 구현할 때의 domain의 predicates는 다음과 같다.

```
(:predicates
    (in ?water ?glass)
    (is ?obj ?level)
    (color ?water ?col)
    (on ?water ?underwater)
    (top ?water)
    (free ?water)
)
```

- ?water는 ?glass 안에 있다
- ?obj은 ?level의 레벨이다
- ?water는 ?color 색깔이다
- ?water밑에 ?underwater 가 있음
- ?water가 유리병의 최상위위치 이다.
- ?water는 더이상 사용되지않는다

초기, ?water가 소속된 ?glass의 위치, 각 ?water와 ?glass의 레벨, ?water의 색깔등으로만 제어를 시도했다. 하지만 위의 조건만으로는 너무 큰 연산이 필요했다. 그래서 최상위 위치, ?water를 free시키는 등의 조건을 추가하였다.
아래는 ?water와 ?glass의 Level의 이해를 돕기위한그림이다.



아래는 구현 초기의 level one water를 옮기는 것의 코드 일부이다.

초기 코드

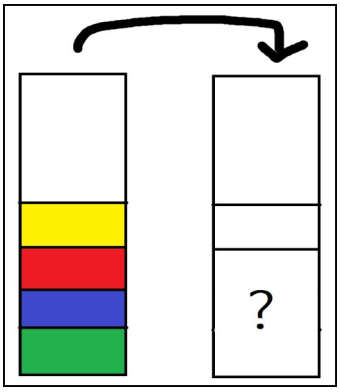
```
(define (domain watersortpuzzle)
(:requirements :strips :equality :adl :fluents)
(:types glass water)
(:predicates
    (in ?water ?glass)
    (is ?obj ?level)
    (color ?water ?col)
    (on ?water ?underwater)
    (istop ?glass ?water)
    (free ?water)
)
(:action pour1
    :parameters (?glass1 ?glass2 - glass
                 ?underwater1 ?water1 ?water2 - water
    )
    :precondition (and
        (not(free ?water1)) (not(free ?water2)) (not(free ?underwater1))
        (istop ?glass1 ?water1)
        (is ?water1 one)
        (or(istop ?glass2 ?water2)
            (is ?glass2 zero)
        )
        (or (and(color ?water1 red)(color ?water2 red))
            (or (and(color ?water1 yellow)(color ?water2 yellow))
                (or (and(color ?water1 blue)(color ?water2 blue))
                    (or (and(color ?water1 green)(color ?water2 green))
                        (is ?glass2 zero)
                    )
                )
            )
        )
        (or(on ?water1 ?underwater1)
        (is ?glass1 one)
        )
        (not (is ?glass2 four))
    )
```

```
:effect (and
    (when
        (and (is ?glass1 one))

        (and (is ?glass1 zero)
        (not (istop ?glass1 ?water1))
        (not (is ?glass1 one))
        )
    )
    (when
        (and (is ?glass1 two))

        (and (is ?glass1 one)
        (not(is ?glass1 two))
        (istop ?glass1 ?underwater1)
        (not (istop ?glass1 ?water1))
        (not(on ?water1 ?underwater1))
        )
    )
    (when
        (and (is ?glass1 three))

        (and (is ?glass1 two)
        (not (is ?glass1 three))
        (istop ?glass1 ?underwater1)
        (not (istop ?glass1 ?water1))
        (not (on ?water1 ?underwater1))
        )
    )
    (when
        (and (is ?glass1 four))

        (and (is ?glass1 three)
        (not (is ?glass1 four))
        (istop ?glass1 ?underwater1)
        (not (istop ?glass1 ?water1))
        (not (on ?water1 ?underwater1))
        )
    )
```

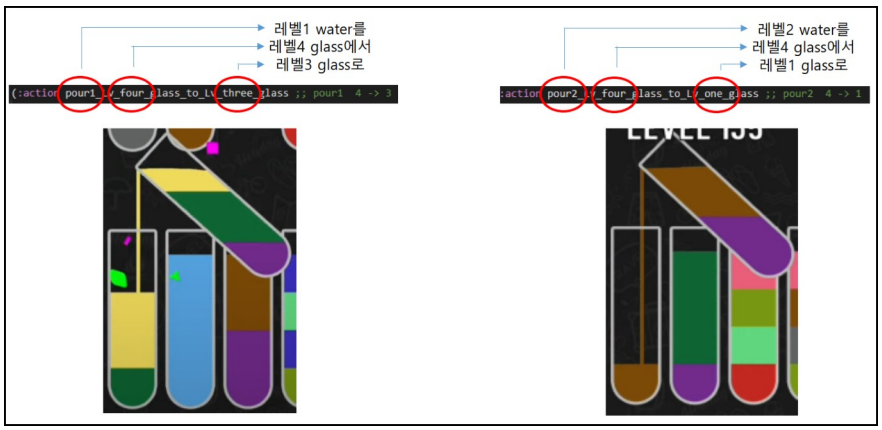
```
(when
    (and
        (not (is ?glass2 zero))
        (is ?water2 one)
    )

    (and
        (is ?water2 two)
        (free ?water1)
    )
)
(when
    (and
        (not(is ?glass2 zero))
        (is ?water2 two)
    )

    (and
        (is ?water2 three)
        (free ?water1)
    )
)
(when
    (and
        (not(is ?glass2 zero))
        (is ?water2 three)
    )

    (and
        (is ?water2 four)
        (free ?water1)
    )
)
(when
    (and
        (is ?glass2 zero)
    )
```

위 코드를 간단히 설명하면, action의 precondition에서 알맞은 glass를 선택하였는지 (각 glass의 water color가 같은지, glass혹은 water의 level이 알맞은

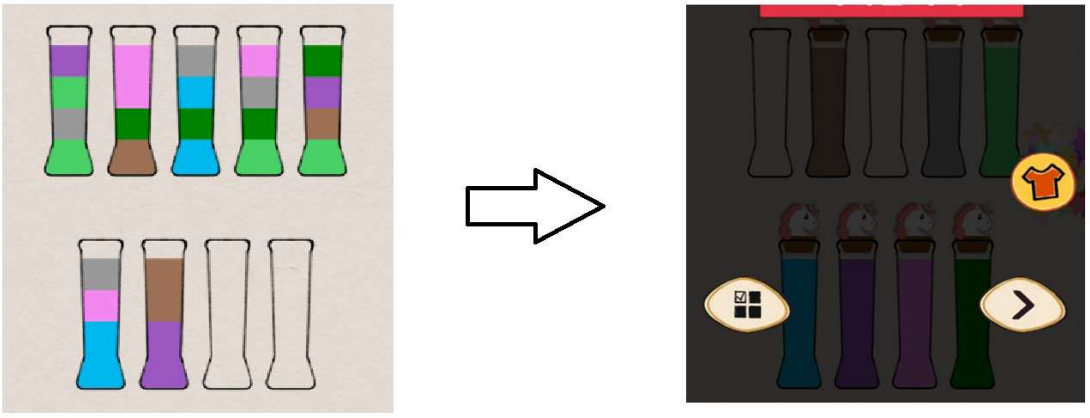
지 등)를 검사한 뒤 effect에서 when(effect에서 한번더 condition과 effect를 부여하는 것. if와 비슷)을 사용하여 각 glass와 water의 현재 상태에 따라 effect를 추가결정하는 구조이다. 추가적으로 각 glass와 water의 현재상태에 따라 effect 가 달라져야 하는 이유는 다음과같다.



?glass level 4 에서 ?glass level 3으로 ?water level 1 을 옮긴다고 가정하였을때, ?glass2의 top에 있는 ?water의 레벨이 2일때는 ?water의 레벨이 3이 될것이고, top에 있는 ?water의 레벨이 3이면 ?water의 레벨은 4가 될 것이다. 이와 비슷한 예시로 ?glass level 0 으로 옮기게 된다면 ?glass level 4 에서 옮겨진 ?water는 free시키지 않아야 하는등의 effect가 달라지게 된다. (숫자를 대입하여 비교를 하게되면 위와 비슷한 조건들이 완화되게 된다. ?glass level이 0보다 큰 ?glass로 옮긴다면, ?glass의 level을 +1시키는등으로 effect 부분의 코드가 줄어들게 된다. 프로젝트 진행 초반 최초 코드에서는 Numeric Fluents를 사용하여 숫자를 대입하여 코드를 작성하였지만 연산시간과 메모리 관련 오류로 코드에 오류가 없다는 입증이 힘들었다.) 하지만 위의 코드로는 연산량의 문제(해당 문제로 여러 실험을 거쳤다. 플래닝 엔진이 어떤구조로 연산을 실행하는지는 뜯어보지 못했기 때문에 정확한 이유는 모르지만 effect에서의 연산문제가 있는 것 같았다. effect를 한줄씩 지우며 확인해본결과 특정 수의라인에서 조건을 검사할 때 problem의 goal과 관계없는 문구를 지워도 plan이 도출되는 결과가 있었다.)로 plan을 도출 할 수 없었다. 그 이유로 action을 세분화 하고 각 aciton에서 검사될 precondition을 더 자세하게 제한했다. action이 세분화되고, precondition에서의 검사가 강해지면 effect에서의 연산량이 작아져서 기존코드보다 더 큰 규모의 plan을 도출 할 수 있을것이라는 가정이었다. 최종 코드는 다음과같다.(*첨부1)



각 ?water의 레벨과 ?glass의 레벨에 따라 action을 나누었고, ?glass level 1에서 ?glass level 0으로 옮기는 것과 같이 필요없는 action은 삭제시켰다. 그결과 총 action의 수는 25개가 나왔다. 최종코드도 기존코드와 마찬가지로 연산량의 문제로 전체 plan을 도출 하지는 못했다.하지만 action의 세분화로 진행되는 과정마다 plan을 도출(필요햔 action을 제외한 action을 삭제)하여 합치는 것은 가능했다. 아래그림과 같이 ?glass 오브젝트 수가 9이고, ?water 오브젝트 수는 24인 퍼즐의 PLAN을 도출한 뒤, 각각의 action수를 더하여 계산하였을때 PLAN의 단계수는 20이었다.



전체 plan을 도출하기위해서는 굉장히 번거로운 과정을 거쳐야한다. 한번 plan을 도출한 뒤 domain과 problem을 모두 도출된 plan을 적용한 상태에 적합하게 수정해주어야한다. 추가적으로 도출된 plan이 optimal한지 증명하기 위해 해당 워터솔트 퍼즐 스테이지를 직접 반복적으로 클리어 해보았지만 20회가량의 결과에서는 도출된 plan이 가장 optimal하였다. 하지만 action을 나누어서 테스트 하였기 때문에 해당 plan이 optimal한 것을 입증하기는 어려울 것이다.

III.결론

이 프로젝트는 워터솔트 퍼즐 문제를 풀기위한 plan을 제시한다. pddl로 작성된 problem 파일에 워터솔트 퍼즐 문제를 입력시킨 뒤 실행하면 적절한 plan을 제공해준다. 출력된 plan의 순서를 따라 진행하면 해당 워터솔트 퍼즐문제를 최단 plan으로 해결할 수 있다.

이 프로젝트의 플래닝을 이용한 워터솔트 퍼즐에 대한 plan 도출은 성공하였지만, 아직 반쪽짜리 성공이다. 완벽한 성공이 되려면 적절한 플래닝 엔진을 찾거나, pddl이 아닌 다른언어로 작성하는 등 환경을 바꾸는 방법으로 해결할 수 있을 것이다.

pddl로 워터솔트 퍼즐에 대한 플래닝을 설계하는 것은 굉장히 힘들었다. pddl은 domain의 requirements에 라이브러리를 추가하지 않는 이상은 직접 숫자를 대입해 대소를 비교하지 못하고, Numeric Fluents를 사용하여 숫자를 대입해도 duration에 따라 수행되는 (예: 자동차의 연료, 거리, 에너지 등) 것에 특화되어 적용시키기 애매한 부분이 많았다. 기본적으로 모든조건은 true/false 값으로 구분되기때문에 고차원적인 조건이 필요할때는 더 많은 predicates가 필요했다. 그리고 구조체 개념과 비슷하게 glass.level 등의 문법이 가능했다면 구현이 더 쉬웠을 것 이라고 생각한다. 이것들은 컴퓨터가 연산하기는 좋을지 모르겠지만 개발자 입장에서는 불편하다고 느낄 것 같다. 그동안 사용했던 언어들이 얼마나 개발자에게 편한 언어였는지 다시한번 깨닫는 계기가 되었다. 위의 이유로 다시 플래닝을 하게된다면 pddl이 아닌 다른환경에서 설계하게 될 것 같다.

이번 프로젝트로 플래닝에 대한 공부가 많이 되었다. 유니티 AI플래너를 사용하여 워터솔트 퍼즐을 다시 구현하거나 새로운 프로젝트를 진행하는 방향으로 발전시킬 것 이다.

Ⅳ.참고 문헌

1. https://planning.wiki/ref/pddl
(pddl 라이브러리(Requirements)등 참고)

Ⅴ.부록

별첨