BOĞAZIÇI UNIVERSITY

INTRODUCTION TO ARTIFICIAL INTELLIGENCE
CMPE 480

Project 2 Report

Author:
M. Akın Elden
2015402072

7 December 2019

1 Introduction

The domain and problem codes are placed into two separate files. Numbers of cargo, docks or boats are not constrained or the number of cargo types are not constrained. The only assumption is that boat has only one cargo capacity.

2 Domain File

Environment

```
(define (domain cross-river)

(:requirements :strips :typing)

(:types cargo dock boat)
```

Since the goal is crossing the cargoes over river, the domain name is *cross-river*. There are 3 different type of objects in that domain:

- **cargo** : The objects that are carried over river.
- **dock** : The shores of the river.
- **boat** : The boat that carries the objects.

Since the types are determined in the domain file, *:typing* is added to requirements.

Predicates

```
(:predicates
    (cargo-at ?c - cargo ?d - dock)
    (boat-at ?b - boat ?d - dock)
    (free ?b - boat)
    (carrying ?b - boat ?c - cargo )
    (eats ?c1 - cargo ?c2 - cargo)
)
```

There are 5 predicates defined in the domain.

- **cargo-at** ?c ?d : Whether the cargo c is at dock d.
- **boat-at** ?b ?d : Whether the boat b is at dock d.
- **free** ?b : Whether the boat b is free, not carrying any cargo.
- **carrying** ?b ?c : Whether the boat b is carrying cargo c.
- **eats** ?c1 ?c2 : Whether cargo c1 can eat cargo c2.

Actions

```
(:action load-cargo
    :parameters (?b - boat ?c - cargo ?d - dock)
    :precondition (and (boat-at ?b ?d)
                       (cargo-at ?c ?d)
                       (free ?b)
    )
    :effect (and (not (cargo-at ?c ?d))
                 (not (free ?b))
                 (carrying ?b ?c)
    )
)
```

"load-cargo" action takes a cargo (*?c*) from the given dock (*?d*) and loads it to the boat (*?b*). Both the boat and the cargo must be at the given dock and the boat must be free in order to load the cargo. When the action is performed, the cargo is not at dock anymore and the boat is not free. The boat is carrying the cargo.

```
(:action unload-cargo
    :parameters (?b - boat ?c - cargo ?d - dock)
    :precondition (and (boat-at ?b ?d)
                       (carrying ?b ?c)
    )
    :effect (and (not (carrying ?b ?c))
                 (cargo-at ?c ?d)
                 (free ?b)
    )
)
```

"unload-cargo" action is the reverse of "load-cargo" action. It unloads the boat (*?b*) and puts the carried cargo (*?c*) to the given dock (*?d*). Boat must be at the given dock and carrying the given cargo before the action. When the action is performed, the boat is not carrying the cargo anymore, the cargo is at the given dock and the boat is free.

```
(:action move-boat
    :parameters (?b - boat ?from ?to - dock)
    :precondition (and (boat-at ?b ?from)
                 (not (exists (?c1 ?c2 - cargo) (and (eats ?c1 ?c2) (cargo-at ?c1 ?from) (
                     cargo-at ?c2 ?from))))
    )
    :effect (and (not (boat-at ?b ?from))
                 (boat-at ?b ?to)
    )
)
```

"move-boat" action moves boat (*?b*) from one dock (*?from*) to another (*?to*). The boat must be at first dock (*?from*) before the action begins. If there exists two such cargoes (*?c1 ?c2*) that both of

them are left at first dock and one of them eats the other one, then boat cannot move. Important point is that, loaded cargoes are not considered at first dock. Since the boat can move whether it's free or carrying a cargo, it's not added as a predicate. When the action is performed, the boat's location changes. It's not at first dock anymore, it's at second dock (*?to*).

3 Problem File

Environment

```
(define (problem wgc-crossing) (:domain cross-river)

(:objects farmerboat - boat wolf goat cabbage - cargo asia europe - dock)
```

The problem name is *wgc-crossing* since a farmer tries to cross a wolf, a goat and a cabbage over the river. There is a boat object in the problem: **farmerboat**. There are three different cargo objects: **wolf, goat** and **cabbage**. There are two dock objects **asia** and **europe**.

Initial State

```
(:init
    (boat-at farmerboat asia)
    (cargo-at wolf asia)
    (cargo-at goat asia)
    (cargo-at cabbage asia)
    (free farmerboat)
    (eats wolf goat)
    (eats goat cabbage)
)
```

Initially the farmerboat and all the cargoes are at asia dock. The farmerboat is free. The wolf can eat the goat and the goat can eat the cabbage.

Goal State

```
(:goal (and
    (cargo-at wolf europe)
    (cargo-at goat europe)
    (cargo-at cabbage europe)
    (boat-at farmerboat europe)
    )
)
```

Since the farmer wants to cross the river and take other objects with him, we want all the cargoes and the farmerboat to be at europe dock as a goal.

4 Solution

The solution plan found with PDDL solver:

```
(load-cargo farmerboat goat asia)
(move-boat farmerboat asia europe)
(unload-cargo farmerboat goat europe)
(move-boat farmerboat europe asia)
(load-cargo farmerboat cabbage asia)
(move-boat farmerboat asia europe)
(unload-cargo farmerboat cabbage europe)
(load-cargo farmerboat goat europe)
(move-boat farmerboat europe asia)
(unload-cargo farmerboat goat asia)
(load-cargo farmerboat wolf asia)
(move-boat farmerboat asia europe)
(unload-cargo farmerboat wolf europe)
(move-boat farmerboat europe asia)
(load-cargo farmerboat goat asia)
(move-boat farmerboat asia europe)
(unload-cargo farmerboat goat europe)
```