UNIVERSITÀ DEGLI STUDI DI GENOVA

DIBRIS

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY,
BIOENGINEERING, ROBOTICS AND SYSTEM ENGINEERING

ARTIFICIAL INTELLIGENCE FOR ROBOTICS II

First Assignment

Planning

Author:

Carmine Miceli - 5626492
Ecem Isildar - 5430086
Luca Petruzzello - 5673449

Professors:

Mauro Vallati
Fulvio Mastrogiovanni

March 27, 2023

Contents

1 Introduction

The aim of this report is to explain how to model the domain representing an automated coffee shop. After a very brief introduction of the problem, the planning engine and language used, there are sections devoted to explain step by step the use of types, predicates and functions to model all the actions, processes and events needed to solve the problems, including all the optional features requested. Finally, a section analyzes the performance of the planner.

2 Automated coffee shop

The coffee shop has two robots, one barista and one waiter. The barista is at the bar and its role is to prepare cold and warm drinks with different preparation times. When the drinks are ready, the waiter robot can carry them either using its gripper or using a tray. Up to three drinks can be carried using the tray, whereas the waiter's moving speed is reduced in order not to pour out the drinks. The waiter moves at 2 meters per time unit; 1 meter per time unit if it is using the tray. Then, when all the clients have left the tables the waiter needs to clean them, which takes 2 time units per square meter to clean each one; in order to clean the table it needs to leave the tray at the bar. In the environment of the coffee shop, shown in figure (2.1), there are four tables, a bar and an additional place called charge spot designed to identify where each waiter is located at the beginning of the serving routine; concerning the sizes of the table, the third table is 2 square meters, others are 1 square meter. The distances are 1 meter apart from any other, and the bar is 2 meters away from the first two tables.

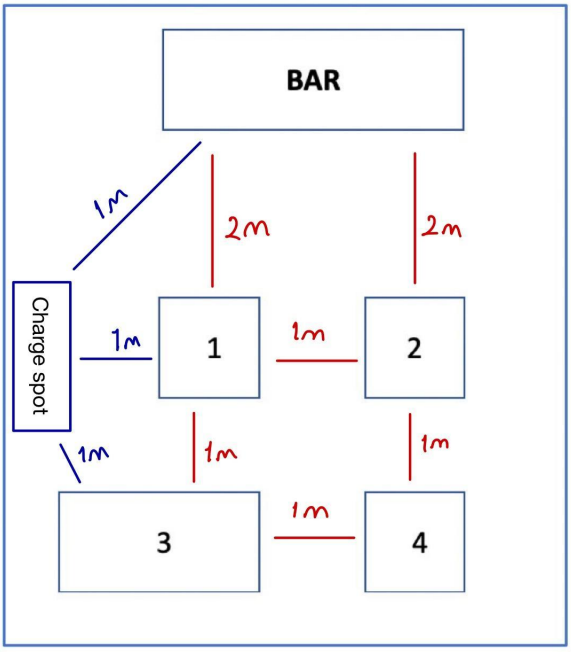

Figure 1: Coffee shop layout

There are four different planning problems to be solved, along with four additional extensions:
1. `Warm drinks cooling down`: The waiter has to serve the warm drinks before they get cold. A warm drink takes 4-time units to become cold, in this case the drink can not be served to the customer.
2. `Two waiters`: Each waiter needs to be assigned to different tables and can just serve that table.
3. `Finishing the drink`: The waiter needs to clean the table when the customers finish their drinks and leave, which takes 4-time units.
4. `Serving food`: The waiter has to serve biscuits to the customers who order cold drinks. Biscuits must be served only after the cold drink has been given.

3 Installing and running

3.1 ENHSP

In order to implement all the features posed by the problems and take into account the time constraints, the planning language chosen is PDDL+ because it is the most expressive of the PDDL family; due to this choice, ENHSP was selected as planning engine.

ENHSP, Expressive Numeric Heuristic Search Planner, is a PDDL automated planning system that supports:

- *Classical and Numeric Planning (PDDL2.1)*
- *Optimal Simple Numeric Planning*
- *Satisficing Non-Linear Numeric Planning*
- *Planning with Autonomous Processes and Discrete Events (PDDL+)*
- *Global Constraints and Expressive Formulas in Preconditions and Goals*

We used the default setting of the engine to find a plan for each problem (the engine discretizes each problem with a $\delta = 1.0$ and the algorithm used is Greedy Best First Search).

3.2 Installing

The engine is written in JAVA and works well with JAVA 15. Use the following command to obtain it:

sudo apt-get install openjdk-15

Then go to the GitLab page, switch to the enhsp-20 branch and clone the repository.

The last step is to run *./compile* in the root folder.

3.3 Running

Use the following command in order to execute the planner:

java -jar enhsp-dist/enhsp.jar -o <domain-file> -f <problem-file>

4 Design of the domain

The design of the model was carried out in a general manner by taking into account all the requirements posed by the problem, so that the implementation of the options did not affect the underlying base.
The choice of EHNSP as planning engine limited the possibility of using durative actions, which were turned into *action - event - process*, thanks to the richness of the planning language adopted: PDDL+.
In order to deal with the objects of the problem, some minimal types and also supertypes were used to avoid duplicating predicates or actions, from these choices the predicates were then derived, in details:

- ***barista***
- ***waiter***
- ***client***
- ***warm cold - drink***
- ***gripper***
- ***table bar station - location***
- ***tray***
- ***client***

Positional predicates that define where an object is located:

- *(at-rob ?r - waiter ?l - location)*
- *(at-client ?c - client ?l - table)*
- *(at-drink ?d - drink ?l - location)*
- *(at-tray ?t - tray ?l - bar)*

State predicates concerning the waiter and some of its features (e.g. gripper):

- *(empty ?g - gripper ?w - waiter)*
- *(hold-drink ?g - gripper ?d - drink ?w - waiter)*
- *(hold-tray ?g - gripper ?t - tray ?w - waiter)*
- *(fast-moving ?w - waiter ?from ?to - location)*
- *(slow-moving ?w - waiter ?from ?to - location)*
- *(waiter-cleaning ?w - waiter)*

State predicates concerning drinks and clients:

- *(ready ?d - drink)*
- *(on-tray ?t - tray ?d - drink)*
- *(request ?d - drink ?c - client)*
- *(served ?d - drink ?c - client)*

State predicates concerning tables:

- *(dirty ?t - table)*
- *(table-cleaning ?t - table)*
- *(clean ?t - table)*

In order to take into account the numerical aspects of the problem, e.g. the size of the tables, several functions were created and used.
Functions concerning the time for the waiter:

- *(time-moving ?w - waiter)* - time elapsed while moving to reach a location
- *(time-cleaning ?w - waiter)* - time elapsed while cleaning the tables

Functions concerning the drinks:

- *(ready-time ?d - drink)* - time needed to prepare the drink
- *(time-prepared ?d - drink)* - time elapsed from the preparation of the drink

Functions concerning the table:

- *(size-table ?t - table)*
- *(client-for-table ?t - table)*

Function concerning the time needed for a client to drink:

- *(time-to-drink ?c - client)*

Function to take into account the distance between different locations:

- *(distance ?l1 ?l2 - location)*

Function to count how many drinks have been put on the tray:

- *(tray-level ?t - tray)*

In the following sections, the solution to the problem will be explained by following the logic order behind the actions needed in order to achieve the goal.

4.1 Barista prepares the drinks

The **(prepare-warm-drink)** action starts the **(making-warm-drink)** process and has three preconditions and three effects. The first precondition **(free ?a)** indicates that the bartender is not currently making a drink; the second precondition **(request ?wd ?c)** associates the drink with the client; the third precondition **(not (ready ?wd))** indicates that the drink is not currently ready. The first effect is that the drink is at the bar **(at-drink ?wd ?b)**; the second effect is that the bartender starts to prepare the drink and it is no longer free to prepare others; the third effect is to assign zero to a function, **(assign (ready-time ?wd) 0)**, which is necessary for the time simulation of the preparation process. These effects trigger the **(making-warm-drink)** process.
This process has the task of increasing the timer function and simulating the timing required for making the warm drink, it has three preconditions and one effect. The first precondition is that the drink is at the bar **(at-drink ?wd ?b)**; the second precondition is that the bartender is busy with the drink **(not (free ?a))**; the third precondition is that the timer value is less than 5 **(< (ready-time ?wd) 5)**. The only effect is to increase the previously mentioned timer by 1 for each time unit **(< (increase (ready-time ?wd) (* 1 #t))**.
The **(ready-warm-drink)** event makes the drink ready and has two preconditions and four effects. The first precondition is that the timer must be greater than or equal to 5 **(>= (ready-time ?wd) 5)**; the second precondition is that the drink is not yet ready **(not (ready ?wd))**. The first effect is that the bartender is again available **(free ?a)**; the second effect is needed to invalidate the event itself and to indicate that the drink is ready **(ready ?wd)**; the third effect is that the cooling predicate is assigned to the drink which is essential for the simulation of the cooling of the drink **(cooling ?wd)**; the fourth effect initialises a timer at zero which is essential for delivering the drink within four time units **(assign (time-prepared ?wd) 0)**.
As far as cold drinks are concerned, the mode of action is the same, in fact the **(prepare-cold-drink)** action, the **(making-cold-drink)** process and the **(ready-cold-drink)** event have the same structure as the ones above. The only differences are that in the **(making-cold-drink)** process the precondition on the timer states that this must be less than 3 **(< (ready-time ?cd) 3)**, in the **(ready-cold-drink)** event the precondition on the timer states that this must be greater than or equal to 3 and in the effects we do not find **(cooling ?wd)** and **(assign (time-prepared ?wd) 0)**.
It is important to remember that for warm and cold drinks we use two different types (wd and cd) derived from the drink supertype.

4.2 Waiter takes and gives drinks

For taking the drinks, two actions are available to the waiter: **(take-drink-gripper)** and **(take-drink-tray)**. The first preconditions are the drink is prepared **(ready ?d - drink)** and also the waiter should be at the bar using the predicate **(at-rob ?r - waiter ?l - location)** with holding the drink with the gripper **(hold-drink ?g- gripper ?d - drink ?w - waiter)** or without holding the tray **(hold-tray ?g - gripper ?t - tray ?w - waiter)**, this is a precondition of action take-tray. If it decides to take drinks on the tray **(on-tray ?t - tray ?d - drink)**, the number of drinks on the tray is counted until three using the function **(tray-level ?t - tray)**. Then, it can start moving, either slowly or fast depending on its decision of using the tray or gripper with the help of these preconditions **(fast-moving ?w - waiter ?from ?to - location)**, **(slow-moving ?w - waiter ?from ?to - location)** inside the processes move-slow/fast. For giving the drinks, the waiter needs to be at the table, and also the customer should be still waiting at the table. So, it can serve the drink to the customer. In this case, if the tray is used in the serving process again, the drink that is served is counted in order to understand how many drinks are still on the tray. After finishing all the drinks on the tray, it leaves the tray on the bar with the help of action **(give-tray)**.

4.3 Waiter's motions

Each waiter can move fast if it does not carry the tray, but if it does then it will move slow.
The **(init-fast-motion)** action has the task of starting the **(move-fast)** process, it has four preconditions and two effects. The first precondition is that the waiter is in a specific starting point **(at-rob ?w ?from)**; the second precondition is that the waiter does not hold the tray **(not (hold-tray ?g ?t ?w))**; the third precondition is useful for associating a gripper with the corresponding waiter **(belong ?g ?w))**; the last precondition is needed to indicate that the waiter is not involved in any other action at the present moment, e.g. cleaning the table **(not (waiter-cleaning ?w))**.
The first effect is that the waiter has left the initial location **(not (at-rob ?w ?from))**; the other effect is the use of the predicate **(fast-moving ?w ?from ?to)** to uniquely indicate the state of the waiter and which is fundamental for the **(move-fast)** process. This process has the task of incrementing the timer to simulate the time it takes to reach a place, it has two preconditions and one effect.

The first precondition is that the predicate **(fast-moving ?w ?from ?to)** has been set to true; the second precondition is that the time elapsed since moving **(time-moving ?w)** is less than the mathematical division between the distance of the starting and finishing point **(distance ?from ?to)** and the speed, which for the fast motion is equal to 2 **(< (time-moving ?w) (/ (distance ?from ?to) 2))**, this mathematical expression is clearly the time needed to move from a certain location to another. The only effect is an increase of 1 in the timer that keeps track of the time of movement **(increase (time-moving ?w) (* 1 #t))**.
The **(stop-fast-motion)** event is essential to indicate the waiter's reaching the location and has two preconditions and three effects. The first precondition is that, as before, the predicate **(fast-moving ?w ?from ?to)** has been set to true; the last precondition is that the time elapsed since moving is greater than or equal to the mathematical division between the distance of the starting and finishing point and the speed, **(>= (time-moving ?w) (/ (distance ?from ?to) 2)**.
The first effect is that the waiter is at the destination **(at-rob ?w ?to)**; the second effect is that now the predicate indicating the need for fast-moving is no longer true **(not (fast-moving ?w ?from ?to))**; the last effect is that the timer is set to zero again **(assign (time-moving ?w) 0)**.
The logic behind the slow motion is the same as described so far but with some differences. The different precondition of the **(init-slow-motion)** action concern the waiter holding the tray **(hold-tray ?g ?t ?w)**, whereas in the effects in this case it is necessary to indicate that it is a slow motion **(slow-moving ?w ?from ?to)** which starts the **(move-slow)** process.
This process in the preconditions has, of course, the predicate **(slow-moving ?w ?from ?to)** and that the time elapsed since moving **(time-moving ?w)** is less than the mathematical division between the distance of the starting and finishing point **(distance ?from ?to)** and the speed, which for the slow motion is equal to 1 **(< (time-moving ?w) (/ (distance ?from ?to) 1))**; the effect is to increase the timer **(increase (time-moving ?w) (* 1 #t))**.
The **(stop-slow-motion)** event has the following preconditions **(slow-moving ?w ?from ?to)** and the same control on time elapsed as before, **(>= (time-moving ?w) (/ (distance ?from ?to) 1)**.
The effects are that the waiter is actually at the end position **(at-rob ?w ?to)**, that slow-moving is no longer true **(not (slow-moving ?w ?from ?to))** and that the timer is still initialised to zero **(assign (time-moving ?w) 0))**.

4.4 Waiter cleans tables

In order to clean the tables, the waiter performs first an action called **(start-cleaning)** with parameters: table, gripper and waiter.
The preconditions concern some features of the waiter, for example to be at the table - **(at-rob ?w ?t)**, not involved in any cleaning - **(not (waiter-cleaning ?w))** and the gripper to be empty - **(empty ?g ?w)**; the table to be dirty, without any client and not being under cleaning: **(dirty ?t)**, **(= (client-for-table ?t) 0)** and **(not (table-cleaning ?t))**.
The effects are to uniquely state that the table is under cleaning and the waiter is performing this operation, this is needed in order for the waiter to not be involved in any other actions in the meantime: **(table-cleaning ?t)** and **(waiter-cleaning ?w)**.
The predicates for the effects of the action, set to true, turn to be the preconditions to start a process called **(clean-table)**, which is responsible to simulate the elapse of time required; its parameters are: table and waiter.
An additional precondition is needed with the use of a function in order to measure the time needed to clean the table: **(< (time-cleaning ?w) (* (size-table ?t) 2))**
At each iteration for time unit of the process, the effect is to increase the time spent cleaning for the waiter: **(increase (time-cleaning ?w) (* 1 #t))**
An event called **(done-cleaning)** is used to uniquely state that a specific table does not need to be cleaned anymore, its parameters are: table and waiter.
The preconditions concern the state of the table and the waiter:**(dirty ?t)**, **(waiter-cleaning ?w)**, **(table-cleaning ?t)** and the final time elapsed **(>= (time-cleaning ?w) (* (size-table ?t) 2))**.
The effects must be considered in order to invalidate the event itself, the table is not under cleaning and the waiter is not involved in this activity, therefore it is free to start other actions: **(not (waiter-cleaning ?w))** and **(not (table-cleaning ?t))**; whereas for the state of the table we have: **(not (dirty ?t))** and **(clean ?t)**. Last but not least, the time assigned for cleaning is set to zero **(assign (time-cleaning ?w) 0)** and the number of clients is set to -1 **(assign (client-for-table ?t) -1))** to invalidate the event **(table-dirty)**.

4.5 Optional extensions

To the underlying base of the model explained up to this point, the following predicates were added in order to build the actions needed to implement the required options.
Predicates concerning the features of the waiter:

- *(belong ?g - gripper ?w - waiter)* - each gripper is specified for a waiter
- *(order ?w - waiter ?t - table)* - each waiter is assigned to a specific table
- *(carrying-biscuit ?g - gripper ?w - waiter)* - each waiter is uniquely responsible of carrying biscuits to clients

Predicate to indicate that a warm drink has begun to cool down and must be served: *(cooling ?wd - warm)*.
Predicate to indicate that a specific client, whose drink is cold, has received a biscuit: *(biscuit-given ?c - client)*.

4.5.1 Cooling down warm drinks

In the **(ready-warm-drink)** event, i.e. when the warm drink is ready, the effects include to set to true the predicate **(cooling ?wd)** for the drink in question and to assign to zero the value of the fucntion **(assign (time-prepared ?wd) 0)**.
When **(cooling ?wd)** is set to true, the **(warm-cool-down)** process starts, it has the sole effect of increasing the aforementioned timer by one **(increase (time-prepared ?w) (* 1 #t))**. This timer is essential because in the **(give-drink-tray)** and **(give-drink-gripper)** actions, by which drinks are served, there is a precondition which states that this timer must always be less than or equal to four **(<= (time-prepared ?d) 4))**. Thanks to this, warm drinks are always served within four time units, allowing waiters to serve them before they get cold.

4.5.2 Two waiters

Two waiters are added as an extension, in order to do this a charge spot is added to the domain as a location type and waiter locations are initialized using **(at-rob wairob1 chargespot)**, **(at-rob wairob2 chargespot)**. The distance between the charge spot to the bar side, first and third table is 1 meter. As it is shown in the figure (2.1). For initialization, the following equations are used **(= (distance chargespot barside) 1)**, **(= (distance chargespot table1) 1)**, **(= (distance chargespot table3) 1)**. So, the waiters can start their day at the charge spot. Then, one of them can go to the bar to take the drinks, and at the same time, the other waiter can go for cleaning. This action decreases the total time of solving the problems. In order to prevent going to the same table for the orders, at least one table the waiter is assigned using the order predicate **(order ?w - waiter ?t - table)**. For instance, in the first problem, the first waiter is assigned to the second table and the same logic is applied to the other problems.

4.5.3 Client leaves the table

When a client has received the drink, a process called **(finish-drink)** begins with parameters: client, drink and table.
The preconditions check that the client is actually located at the table - **(at-client ?c ?t)**, it has received the drink - **(served ?d ?c)** and the time to drink, from the initial value of four time units, is greater than zero - **(> (time-to-drink ?c) 0))**.
At each iteration for time unit of the process, the effect is to decrease the time left for a specific client to drink: **(decrease (time-to-drink ?c) (* 1 #t))**.
When the time left to drink for a client ends, an event called **leave-table** begins with the following parameters: client and table.
The preconditions concern the time spent for drinking for a client - **(= (time-to-drink ?c) 0)** and the position **(at-client ?c ?t)**.
The effects are to diminish the number of clients for that specific table **(decrease (client-for-table ?t) 1)** and state that the client is not present at the table anymore - **(at-client ?c ?t)**.
Because of the process and event implemented above, one last event called **(table-dirty)** is needed in order to indicate to the waiter that the table is now empty and dirty, which means that it must be cleaned; its parameter is: table.
The preconditions regard the number of client located at that table **(= (client-for-table ?t) 0)**, which must result to be zero, and not being dirty **(not (dirty ?t))**.
The only and clear effect is to indicate the specific table as dirty **(dirty ?t)** for the waiter that will then perform the action **(start-cleaning)**.

4.5.4 Give biscuits

The biscuits were assumed to be an infinite amount, therefore no type was created for them. The waiter just needs to perform two elementary actions: **`(take-biscuit)`** and **`(give-biscuit)`**.
The first action has the following parameters: waiter, drink, gripper, bar, client and table; the preconditions concern the position of the waiter - **`(at-rob ?w ?b)`**, the state of the gripper of the waiter - **`(empty ?g ?w)`**, the position of the client **`(at-client ?c ?c)`**, if the cold drink has actually been served to the client - **`(served ?d ?c)`** and the biscuit was not given - **`(not (biscuit-given ?c))`**.
The effects of this action concern the state of the biscuit being carried and the gripper holding the biscuit: **`(carrying-biscuit ?g ?w)`** and **`(empty ?g ?w))`**.
The second action has the following parameters: waiter, gripper, table, client.
The preconditions involve the position of the waiter - **`(at-rob ?w ?t)`**, the state of carrying the biscuit **`(carrying-biscuit ?g ?w)`** and the position of the client **`(at-client ?c ?t)`**.
The effects include to indicate that the biscuit was given to a specific client **`(biscuit-given ?c)`**, the gripper is now empty and usable again - **`(empty ?g ?w)`** and the state of carrying a biscuit is falsified **`(not (carrying-biscuit ?g ?w))`**.

5 Performance analysis of ENHSP

In order to evaluate the performances of the planning engine, the problems were tested with different search strategies and heuristics configurations of the planning engine. In order to exploit all the available options, it is enough to add the flag `-planner` with the desired combination of search strategy and heuristic, which can be found here.
The planning engine was tested on a machine equipped with AMD Ryzen 3 with 4 core, 8 GB of memory, running Ubuntu 22; 1 GB of memory was used for every problem.
The obtained results are shown in the following table:

Problem 1				
Engine configuration	Grounding time	Planning Time [ms]	Elapsed Time [s]	Expanded Nodes
sat-hmrp	204	1613	27	136
sat-hmrphj	194	/	/	/
sat-hadd	204	3713	23	7508
sat-aibr	253	4712	23	707
opt-hmax	219	66722	14	461171
opt-hrmax	228	59992	14	461171
opt-blind	249	114490	13	2071641

Problem 2				
Engine configuration	Grounding time	Planning Time [ms]	Elapsed Time [s]	Expanded Nodes
sat-hmrp	241	3815	57	12314
sat-hmrphj	242	/	/	/
sat-hadd	255	105673	44	828828
sat-aibr	240	355174	46	81783
opt-hmax	202	//	//	//
opt-hrmax	171	//	//	//
opt-blind	228	//	//	//

Problem 3				
Engine configuration	Grounding time	Planning Time [ms]	Elapsed Time [s]	Expanded Nodes
sat-hmrp	236	3409	61	1482
sat-hmrphj	233	/	/	/
sat-hadd	215	3758	47	3788
sat-aibr	199	//	//	//
opt-hmax	204	//	//	//
opt-hrmax	192	//	//	//
opt-blind	190	//	//	//

Problem 4				
Engine configuration	Grounding time	Planning Time [ms]	Elapsed Time [s]	Expanded Nodes
sat-hmrp	321	//	//	//
sat-hmrphj	419	/	/	/
sat-hadd	375	5498	83	5170
sat-aibr	308	//	//	//
opt-hmax	336	//	//	//
opt-hrmax	329	//	//	//
opt-blind	366	//	//	//

There are two different types of possible configuration for the planning engine, the first one has the prefix `sat-` and it is designed to work for sat planning. The search strategy used for this kind of configuration is Greedy Best First Search (GBFS), except for sat-aibr which uses A*. The latter is also used in the second type of configuration, which has the prefix `opt-` and it is used for optimal planning.
In the `sat-hmrphj` configuration, a JAVA error was raised which did not make possible to retrieve any data for the problems (notation /).
In some cases, it was not possible to retrieve any data because the planning engine used all the amount of memory available in the machine and did not finish the research for a plan (notation //).
It can be observed that the grounding time is higher as the complexity of the problem increases, this is because the number of variables that need to be substituted with the objects increases. Furthermore, sat planning will take less time to find a plan than the optimal one, this is because it will expand less nodes but it will produce an Elapsed Time for the output plan much higher. In general, it can be said that if the Planning Time is not a high priority it would be preferable to use configurations of the engine that reduce the Elapsed Time, in addition to this a machine with a more performing hardware would improve the performances of the planning engine in finding the plans.