

Automated Planning - Project 1

Simone Nicosanti

February 2024

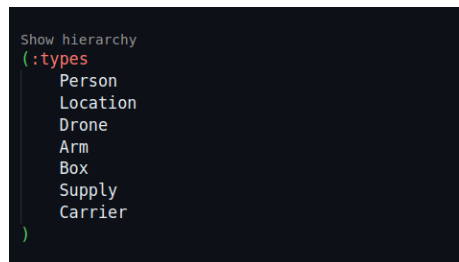
1 Part 1

1.1 Exercise 1 - Model the problem

In the first place, we can analyze the domain.

As for the types they are shown in figure 1. As the figure makes clear we have:

- The *Supply* type has been introduced in order to make the problem model more flexible and not to hard code the supported supply types
- The *Arm* type has been introduced in order to have more flexibility in the number of arms that a drone can have and to have a simpler modeling of the problem, i.e. not having a different action for each arm
- The *Drone* type has been introduced in order to support more than one drone
- The *Carrier* type has been introduced in this part of the assignment but will not be used until *Part 2*



```
Show hierarchy
(:types
  Person
  Location
  Drone
  Arm
  Box
  Supply
  Carrier
)
```

Figure 1: Part 1 - Types

Now we can focus on the predicates shown in figure 2. We can detail each of them:

- *isPersonInLocation* is used to mark the location where a person is

- *personHas* is used to mark whether a person has a type of supply
- *isDroneInLocation* is used to mark the current location of a drone
- *isDroneArmFree* is used to mark whether an arm of a drone is free
- *droneArmHasBox* is used to memorize whether an arm of a drone is carrying a box
- *boxContains* is used to mark which kind of supply a box contains: the introduction of this kind of variable together with the supply type makes the problem more flexible, because we do not have a type for each supply, but we have the type *Box* paired with the type *Supply* so we can add supplies from the problem definition without changing the domain
- *isBoxInLocation* is used to mark the current location of a box

```
(:predicates
  (isPersonInLocation ?p - Person ?l - Location) 1
  (personHas ?p - Person ?s - Supply) 1

  (isDroneInLocation ?d - Drone ?l - Location) 3
  (isDroneArmFree ?d - Drone ?a - Arm) 1
  (droneArmHasBox ?d - Drone ?a - Arm ?b - Box) 1

  (boxContains ?b - Box ?s - Supply) 1
  (isBoxInLocation ?b - Box ?l - Location) 1

  (locationIsDeposit ?l - Location)
)
```

Figure 2: Part 1 - Predicates

Now we can focus on the defined actions; the domain has been designed with only three actions:

- *MoveDrone* (Figure 3a). This action takes as parameter the drone we want to move and the start and end location and simply changes the predicates in order to achieve the drone move.
- *PickBox* (Figure 3b).
 - Parameters. As for the parameters we have:
 - * Drone. Which drone is taking the box
 - * Arm. With which arm the drone is picking up the box. Notice that the introduction of the type *Arm* simplify this action: otherwise we would have an action for both left and right arm of the drone
 - * Box. The box we are picking up
 - * Location. Where we are taking the box

- Preconditions
 - * We have to check that both drone and box are in the same location
 - * We have to check that the arm with which the drone is picking up the box is free
 - * We do not have to check if the arm is owned by the drone: when we write the problem we initialize as *True* the *isDroneArmFree* variable only if the drone owns that arm and, as a consequence, changing both the *isDroneArmFree* and *droneArmHasBox* variable in a consistent way, we do not have to introduce an other variable and precondition.
- Effects. The effects are that:
 - * The arm is no longer free
 - * The box is no longer in the location where it has been picked up
 - * The drone arm now has the picked up box
- *DeliverBox* (Figure 3c).
 - Parameters
 - * Drone, which drone is delivering the box
 - * Arm, which is the arm carrying the box
 - * Box, which box has to be delivered
 - * Supply, which is the supply inside the box
 - * Person, who is going to receive the box
 - * Location, where the person is located
 - Precondition
 - * We have to check that both drone and person are in the same place
 - * We have to check that the arm of the drone really has the box
 - * We have to check that the box contains the supply that is going to be delivered; otherwise we could deliver a box containing something but marking as delivered an other kind of supply
 - Effects
 - * The drone arm now is free, i.e. we mark as *True* the *isDroneArmFree* and as *False* the *droneArmHasBox*
 - * The person has received the supply, so we mark the *personHas* variable as *True*
 - * We did not change the box location and this is because once the box has been delivered to a person it can not be delivered again, so it is *out* of the problem: this is a way to mark the box as delivered, but an other way would have been to introduce an other variable like *boxHasBeenDelivered* and check it in the *PickBox* action, not to pick a box that was already been delivered, and mark it as true in the *DeliverBox* action

```

(:action MoveDrone
:parameters (?d - Drone ?from - Location ?to - Location)
:precondition (and
  (isDroneInLocation ?d ?from)
)
:effect (and
  (not (isDroneInLocation ?d ?from))
  (isDroneInLocation ?d ?to)
)
)

```

(a) Part 1 - MoveDrone

```

(:action PickBox
:parameters (?d - Drone ?a - Arm ?b - Box ?l - Location)
:precondition (and
  (isDroneInLocation ?d ?l)
  (isBoxInLocation ?b ?l)
  (isDroneArmFree ?d ?a)
  ; To not limite drone in picking up boxes only in the deposit
  ; (locationIsDeposit ?l)
)
:effect (and
  (not (isDroneArmFree ?d ?a))
  (not (isBoxInLocation ?b ?l))
  (droneArmHasBox ?d ?a ?b)
)
)

```

(b) Part 1 - PickBox

```

(:action DeliverBox
:parameters (?d - Drone ?a - Arm ?b - Box ?s - Supply ?p - Person ?l - Location)
:precondition (and
  (isDroneInLocation ?d ?l)
  (isPersonInLocation ?p ?l)
  (droneArmHasBox ?d ?a ?b)
  (boxContains ?b ?s)
)
:effect (and
  (not (droneArmHasBox ?d ?a ?b))
  (isDroneArmFree ?d ?a)
  (personHas ?p ?s)
  ; We make the box disappear from the problem
  ; (isBoxInLocation ?b ?l)
)
)

```

(c) Part 1 - DeliverBox

Figure 3: Part 1 - Actions

1.2 Exercise 2 - Problem Generator

We can write a script in order to generate problems of different dimensions without writing them by hand.

In the first part of the generator we have two types of supply defined in the problem which are *food* and *medicine*, while the random seed has been set to 0 to have more predictability and make the generator produce the same problem for each execution (figure 4).

```

# Crates will have different contents, such as food and medicine.
# You can change this to generate other contents if you want.

content_types = ["food", "medicine"]

#####
# Random seed
#####

# Set seed to 0 if you want more predictability...
random.seed(0);

```

Figure 4: Part 2 - Generator Begin

In the first part of the generator we have a parsing of the parameters (figure 5) which are:

- *-d*, the number of drones in the problem
- *-r*, the number of carriers in the problem (in this section it will be consid-

ered as zero)

- $-l$, the number of locations
- $-p$, how many people are in the problem
- $-c$, how many boxes are in the problem
- $-g$, how many bokes are going to be assigned in the goal

```
def main():
    # Take in all arguments and print them to standard output

    parser = OptionParser(usage='python generator.py [-help] options...')
    parser.add_option('-d', '--drones', metavar='NUM', dest='drones', action='store', type=int, help='the number of drones')
    parser.add_option('-r', '--carriers', metavar='NUM', type=int, dest='carriers',
        help='the number of carriers, for later labs; use 0 for no carriers')
    parser.add_option('-l', '--locations', metavar='NUM', type=int, dest='locations',
        help='the number of locations apart from the depot')
    parser.add_option('-p', '--persons', metavar='NUM', type=int, dest='persons', help='the number of persons')
    parser.add_option('-c', '--crates', metavar='NUM', type=int, dest='crates', help='the number of crates available')
    parser.add_option('-g', '--goals', metavar='NUM', type=int, dest='goals',
        help='the number of crates assigned in the goal')
```

Figure 5: Part 2 - Generator parsing parameters

After parsing the parameters, some lists with different objects are initialized (figure 6); the global variable *ARMS_PER_DRONE* has been initialized as 2.

```
drone = []
person = []
crate = []
carrier = []
location = []
arm = []

location.append("depot")
for x in range(options.locations):
    location.append("loc" + str(x + 1))
for x in range(options.drones):
    drone.append("drone" + str(x + 1))
for x in range(options.carriers):
    carrier.append("carrier" + str(x + 1))
for x in range(options.persons):
    person.append("person" + str(x + 1))
for x in range(options.crates):
    crate.append("crate" + str(x + 1))
for x in range(0, ARMS_PER_DRONE * options.drones):
    arm.append("arm" + str(x + 1))
```

Figure 6: Part 1 - Generator Objects List

We can now discuss the three functions that randomizes the distribution of objects in the problem.

First of all we have the *setup_content_types* (figure 7); let us detail it:

1. In the first *for* cycle it is extracted for the first $n - 1$ types of supply is extracted a random number f boxes containing them, while the last supply has the remaining number of boxes containing it.
2. It is then computed the maximum number of goals that can be managed with the current randomization and compared with the number of goals given as parameter

```

def setup_content_types(options):
    while True:
        num_crates_with_contents = []
        crates_left = options.crates
        for x in range(len(content_types) - 1):
            types_after_this = len(content_types) - x - 1
            max_now = crates_left - types_after_this
            print("HELLO", x, types_after_this, crates_left, len(content_types), max_now)
            num = random.randint(1, max_now)
            print(num)
            num_crates_with_contents.append(num)
            crates_left -= num
        num_crates_with_contents.append(crates_left)
        # print(num_crates_with_contents)

        # If we have 10 medicine and 4 food, with 7 people,
        # we can support at most 7*4=11 goals.
        maxgoals = sum(min(num_crates, options.persons) for num_crates in num_crates_with_contents)

        # Check if the randomization supports the number of goals we want to generate.
        # Otherwise, try to randomize again.
        if options.goals <= maxgoals:
            # Done
            break

    print()
    print("Types\tQuantities")
    for x in range(len(num_crates_with_contents)):
        if num_crates_with_contents[x] > 0:
            print(content_types[x] + "\t" + str(num_crates_with_contents[x]))

    crates_with_contents = []
    counter = 1
    for x in range(len(content_types)):
        crates = []
        for y in range(num_crates_with_contents[x]):
            crates.append("crate" + str(counter))
            counter += 1
        crates_with_contents.append(crates)

    return crates_with_contents

```

Figure 7: First Part - Setup Content Type function

3. In the third *for* cycle it is created a list when index *i* is a list of boxes with *i*-th supply

As for the *setup_person_needs* function (figure 8), we have:

1. First of all it is created a matrix of *need*, where *need*[*i*][*j*] is *True* if the *i*-th person needs the supply *j*-th.
2. It is initialized a list of 0 in order to keep track of how many goals have been created for each supply
3. For each goal that we can have according to the goal number given as parameter, we extract a random person and a random supply; if the person still does not have the need set to true for that resource and there are enough boxes for that resource, then we update the variables in a consistent way in order to set the needs.

Finally we have the function *setup_person_location*, which randomizes where people are in the different locations of the problem (figure 9). For each person it is simply decided a random location where to locate it.

After the writing of the header of the problem file, we write in the problem file all the objects (figure 10).

Then we can initialize the state (Figure 11). In detail we have:

- Person

```

# This function generates a random set of goals.
# After you run this, need[personid][contentid] is true if and only if
# the goal is for the person to have a crate with the specified content.
# You will use this to create goal statements in PDDL.
def setup_person_needs(options, crates_with_contents):
    need = [[False for i in range(len(content_types)) for j in range(options.persons)]
            goals_per_contents = [0 for i in range(len(content_types))]

    for goalnum in range(options.goals):
        generated = False
        while not generated:
            rand_person = random.randint(0, options.persons - 1)
            rand_content = random.randint(0, len(content_types) - 1)
            # If we have enough crates with that content
            # and the person doesn't already need that content
            if (goals_per_contents[rand_content] < len(crates_with_contents[rand_content])
                and not need[rand_person][rand_content]):
                need[rand_person][rand_content] = True
                goals_per_contents[rand_content] += 1
                generated = True
    return need

```

Figure 8: First Part - Setup person need

```

def setup_person_location(person : list, location : list) -> list :
    personLocation : list = []

    for per in person :
        locationIndex : int = random.randint(0, len(location) - 1)
        personLocation.append(location[locationIndex])

    return personLocation

```

Figure 9: First Part - Setup Person Location

```

for x in drone:
    f.write("\t" + x + " - Drone\n")

for x in location:
    f.write("\t" + x + " - Location\n")

for x in crate:
    f.write("\t" + x + " - Box\n")

for x in content_types:
    f.write("\t" + x + " - Supply\n")

for x in person:
    f.write("\t" + x + " - Person\n")

for x in carrier:
    f.write("\t" + x + " - Carrier\n")

for x in arm :
    f.write("\t" + x + " - Arm\n")

f.write("\n")

```

Figure 10: First Part - Object Write

- We initialize the location for each person using the variable *isPersonInLocation*
- We initialize what a person has yet using the variable *personNeeds*; this initialization actually is not required as in the goal we will set what person needs and the solver will solve the problem to meet those needs, but we added it for completeness.
- Box
 - We initialize what a box contains using the variable *boxContains* and the previous generated list
 - We initialize what a box contains using the variable *isBoxInLocation*; as default, all boxes are in the *depot* in the first place
- Drone
 - We initialize the drone location using the *isDroneInLocation*; the drone is located in the depot in the first place
 - We initialize the state for each arm of the drone; assuming all drones have the same number of arms, we used a *for* cycle with an `droneIndex` incremented by one every time we have cycled over `ARMS_PER_DRONE` arms.

Finally we can write the goal of the problem (figure 12):

- We make all drones end up in the deposit
- For each person in the problem and for each supply, if the person need the supply (i.e. `need[person][supply] = True`), we add the variable *personHas* for that person and that supply; as a consequence the final goal is that all people defined in the problem have what they actually need.

The generator can be run using the following command:

```
python3 ProblemGenerator.py -d x -r y -l z -p k -c l -g m.
```

Using this generator we can generate problems of increasing size and test them with any solver we want. Let us consider the *ff* solver. For this serie of simulations we fixed number of drones as 1 and carrier number as 0 while increasing the other parameters.

The tests have been done running the planner on each problem for three times and then taking the average of times The biggest problem which *ff* could solve in 60 seconds is shown in table 1. On the other hand, in figure 13 are shown the trends for time and number of actions:

- as the figure 13a makes clear, time has, except for a few outliers, an exponential trend; in dashed black line is shown the time limit
- in figure 13b we can see a linear trend in action number growth over the problem dimension.


```

## Person Locations
for x in range(0, len(person)) :
    f.write(f"isPersonInLocation {person[x]} {person_location[x]}\n")
f.write("\n")

## Person has yet
for i in range(0, len(person)) :
    for j in range(0, len(content_types)) :
        if not need[i][j] :
            f.write(f"personHas {person[i]} {content_types[j]}\n")
f.write("\n")

```

(a) First Part - Person State Init

```

## Box Content
for i in range(0, len(crates_with_contents)) :
    for contBox in crates_with_contents[i] :
        f.write(f"boxContains {contBox} {content_types[i]}\n")
f.write("\n")

## Box Location
for box in crate :
    f.write(f"isBoxInLocation {box} depot)\n")
f.write("\n")

```

(b) First Part - Box State Init

```

## Drone Location
for d in drone :
    f.write(f"isDroneInLocation {d} depot)\n")
f.write("\n")

## Drone Arm State
droneIndex = 0
for armIndex in range(0, len(arm)):
    f.write(f"isDroneArmFree {drone[droneIndex]} {arm[armIndex]}\n")
    if (armIndex != 0 and armIndex % ARMS_PER_DRONE == 0) :
        droneIndex += 1
f.write("\n")

```

(c) First Part -
Drone State Init

Figure 11: First Part - State Initialization

```

f.write("(:goal (and\n")

# All Drones should end up at the depot
for x in drone:
    f.write(f"isDroneInLocation {x} depot)\n")
f.write("\n")

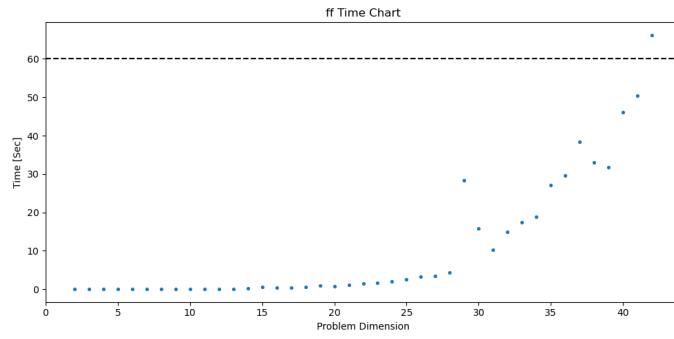
# Each person received what needs
for x in range(options.persons):
    for y in range(len(content_types)):
        if need[x][y]:
            person_name = person[x]
            content_name = content_types[y]
            f.write(f"personHas {person_name} {content_name}\n")
f.write("\n")

```

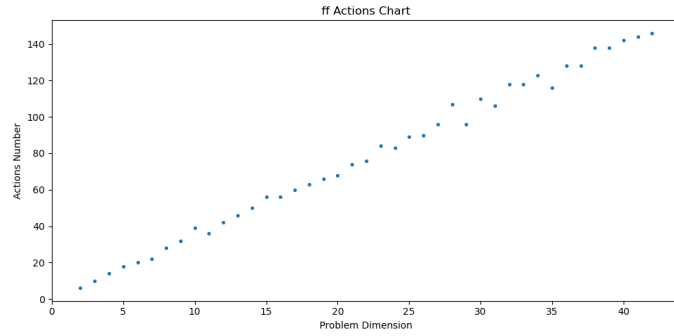
Figure 12: First Part - Goal

Planner	Problem Dimension	Actions Number	Time[s]
ff	41	144	50.419394731521606

Table 1: FF - Biggest Solvable Problem in 60 Seconds



(a) FF - Times over Problem Dimension



(b) FF - Actions over Problem Dimension

Figure 13: FF - Charts

Planner	Problem Dimension	Actions Number	Time [s]
ff	41	144.0	50.419
lpg-td	157	632.0	54.696
sgplan40	96	402.0	41.185

Table 2: Part 1 - Solvers Comparison

1.3 Exercise 3 - Solvers Comparison

We can compare different solvers; let us consider three solvers:

- *ff*
- *sgplan40*
- *lpg-td*

The tests have been done in the same way: three execution on each problem dimension and then we averaged on the results. The biggest problem solvable by each solver is reported in table 2.

On the other hand, in figure 14 are shown charts comparing the solvers in time and number of actions. As we can see in figure 14a all the solvers follow an exponential pattern in time growth, even though *lpg-td* curve grows more slowly than the other two. As for the number of actions, we can see that all three solvers follow a linear pattern, but, as we are considering maximum time of 60 seconds, *ff* and *sgplan40* curves stop before than *lpg-td* curve.

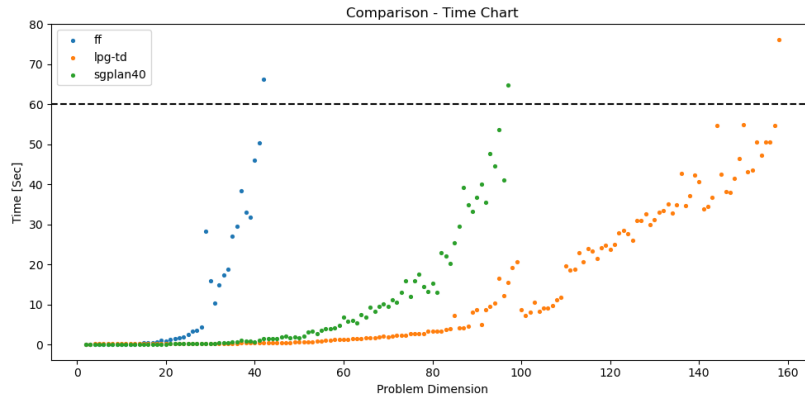
We can compare the time taken by each solver to solve the maximum problem size solvable by the *ff* solver as shown in table 3.

Solver	Problem Dimension	Actions	Time [s]
ff	41	144	50.419
lpg-td	41	166	0.394
sgplan40	41	164	1.029

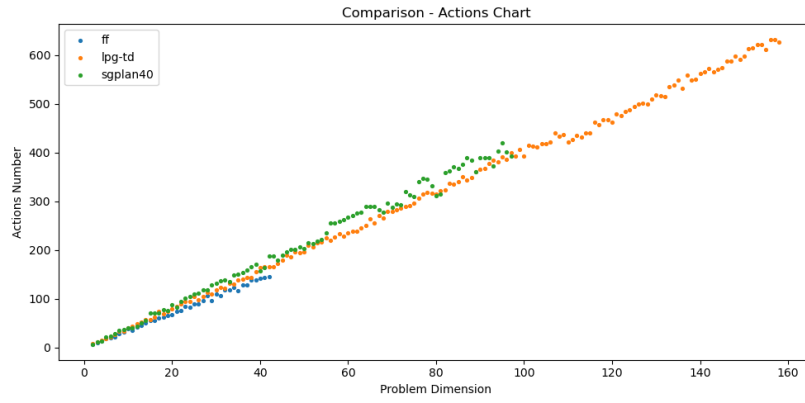
Table 3: Comparison - Biggest FF Problem

In conclusion, as a consequence of what shown in tables and charts, we can say that:

- The best solver in solving big problems is *lpg-td*
- As for solve time the best planner is *lpg-td* as well
- As for number of actions, for small size problem the best solver would be *ff*, while for medium size problems the best solver would be *sgplan40* and for big size problem the best (and only solver) would be *lpg-td*



(a) Comparison - Times



(b) Comparison - Actions

Figure 14: Comparison - Charts

2 Part 2

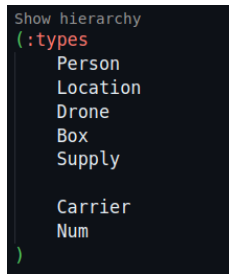
In the second part we introduced the carrier which a drone can use in order to pick up more boxes than its arms number would allow.

2.1 Exercise 1

2.1.1 Modify Domain

First of all we can modify the domain according to the changes made.

As we can see in figure 15, the most of the types are the same as the previous part, but we added the type *Num*: this type is used to define numbers using predicates like explained in the following parts. As the figure shows, we removed the type *Arm*: we want that a drone can pick up the carrier only if all its arms are free, but checking this condition would have been tricky and we decided to remove the type *Arm* making it possible for the drone to carry only one thing between box and carrier.



```
Show hierarchy
(:types
  Person
  Location
  Drone
  Box
  Supply

  Carrier
  Num
)
```

Figure 15: Part 2 - Types

With regard to predicates shown in figure 16:

- *isDroneArmFree* has been replaced with *isDroneFree* as it can only take one load per time
- *droneArmHasBox* has been replaced with *droneHasBox*
- *isNext* is a predicate introduced in order to check whether two number are in a relationship of succession
- *carrierHasBoxes* is a predicate marking how many boxes are in the carrier at a certain time
- *isCarrierInLocation* is a predicate marking the current location of the carrier
- *isBoxInCarrier* is a predicate saving whether a box is or is not in a carrier

As for the actions we have the same for:

```

(:predicates
  (isPersonInLocation ?p - Person ?l - Location) 1
  (personHas ?p - Person ?s - Supply) 1

  (isDroneInLocation ?d - Drone ?l - Location) 6 2 2
  (isDroneFree ?d - Drone) 3 2 2
  (droneHasBox ?d - Drone ?b - Box) 2 2 2

  (boxContains ?b - Box ?s - Supply) 1
  (isBoxInLocation ?b - Box ?l - Location) 1 1

  (locationIsDeposit ?l - Location)

  (isNext ?n1 - Num ?n2 - Num) 2

  (carrierHasBoxes ?c - Carrier ?n - Num) 2 2 2 ; How many boxes
  (isCarrierInLocation ?c - Carrier ?l - Location) 3 1 1 ; Current Location
  (isBoxInCarrier ?c - Carrier ?b - Box) 1 1 1
)

```

Figure 16: Part 2 - Predicates

- *PickBox*
- *MoveDrone*
- *DeliverBox*

With the only difference that the *Arm* type has been removed and the *isDroneArmFree* has been replaced with *isDroneFree* and *droneArmHasBox* has been replaced with *droneHasBox*.

Then we added three additional actions to manage the carrier:

- *MoveCarrier* (figure 17a)
 - Parameters
 - * Drone moving the carrier
 - * Carrier to be moved
 - Preconditions
 - * Carrier and Drone have to be in the same location
 - * The Drone has to be free
 - Effects
 - * Change of location for both Carrier and Drone
- *PutBoxOnCarrier* (figure 17b)
 - Parameters
 - * Drone putting the box on carrier
 - * Carrier where we are putting the box
 - * Location where we are loading the carrier
 - * Box that we are putting on the carrier

- * *StartNum* and *EndNum* to increase the number of boxes that are in the carrier
- Preconditions
 - * We have to check that both Drone and Carrier are in the same location
 - * We have to check that the drone had previously picked up the box
 - * We have to check that on the carrier are *StartNum* boxes
 - * We have to check that *EndNum* is the next of *StartNum*: this is done because otherwise we could go from 1 to 4 boxes in the carrier, while we want to follow the right numerical order
- Effects
 - * The drone is free and it has no boxes
 - * The number of boxes in the carrier moves from *StartNum* to *EndNum* (which, because of the condition on *isNext*, is *StartNum* + 1)
 - * The box is now on the carrier
- *PickBoxFromCarrier* (figure 17c)
 - Parameters. They are the same as *PutBoxOnCarrier*
 - Preconditions
 - * Both drone and carrier are in the same location
 - * The box has to be on the carrier
 - * The drone has to be free
 - * The carrier has *StartNum* boxes in it
 - * The *StartNum* has to be the next of *EndNum*: this is done because otherwise we could go from 4 to 1 boxes in the carrier, while we want to follow the right numerical order; here we have a reverse condition than *PutBoxOnCarrier*, because now the *EndNum* has to be lower than the *StartNum* as we are deloading the carrier
 - Effects
 - * The drone is no longer free and it has the box
 - * The box is no longer in the carrier
 - * The carrier now has *EndNum* boxes (which, as for the condition *isNext*, is *StartNum* – 1)

Notice that when we put the box on a carrier, the drone already has the box and, as a consequence, the *PutBoxOnCarrier* action has to be preceded by a *PickBox* action; on the other hand the *DeliverBox* action can be preceded by both *PickBoxFromCarrier* and *PickBox* action.

```

(:action MoveCarrier
:parameters (?d - Drone ?c - Carrier ?from - Location ?to - Location)
:precondition (and
  (isDroneInLocation ?d ?from)
  (isCarrierInLocation ?c ?from)
  (isDroneFree ?d)
)
:effect (and
  (not (isDroneInLocation ?d ?from))
  (not (isCarrierInLocation ?c ?from))

  (isDroneInLocation ?d ?to)
  (isCarrierInLocation ?c ?to)
)
)

```

(a) Part 2 - Move Carrier

```

(:action PutBoxOnCarrier
:parameters (?d - Drone ?c - Carrier ?l - Location ?b - Box ?startNum - Num ?endNum - Num)
:precondition (and
  (isDroneInLocation ?d ?l)
  (isCarrierInLocation ?c ?l)
  ; (locationIsDeposit ?l)

  (droneHasBox ?d ?b)
  (carrierHasBoxes ?c ?startNum)
  (isNext ?startNum ?endNum)
)
:effect (and
  (not (droneHasBox ?d ?b))
  (isDroneFree ?d)

  (not (carrierHasBoxes ?c ?startNum))
  (carrierHasBoxes ?c ?endNum)
  (isBoxInCarrier ?c ?b)
)
)

```

(b) Part 2 - PutBoxOnCarrier

```

(:action PickBoxFromCarrier
:parameters (?d - Drone ?c - Carrier ?l - Location ?b - Box ?startNum - Num ?endNum - Num)
:precondition (and
  (isDroneInLocation ?d ?l)
  (isCarrierInLocation ?c ?l)
  (isBoxInCarrier ?c ?b)

  (isDroneFree ?d)
  (carrierHasBoxes ?c ?startNum)
  (isNext ?endNum ?startNum)
)
:effect (and
  (not (isDroneFree ?d))
  (droneHasBox ?d ?b)

  (not (carrierHasBoxes ?c ?startNum))
  (not (isBoxInCarrier ?c ?b))
  (carrierHasBoxes ?c ?endNum)
)
)

```

(c) Part 2 - PickBoxFromCarrier

Figure 17: Part 2 - Carrier Actions

2.1.2 Modify Problem Generator

We can change the problem generator in order to support the changes in the domain.

First of all we introduced a new parameter *CarrierSize* to the script in order to support a not hard coded carrier dimension (figure 18).

```
for x in range(options.carrierSize + 1) :  
    nums.append(f"num{x}")
```

Figure 18: Part 2 - Carrier Size

We added the *Num* object as shown in figure 19

```
for x in nums :  
    f.write(f"\t{x} - Num\n")
```

Figure 19: Part 2 - Num object write

Finally we can initialize the state for the new parts as shown in figure 20. In detail:

- For each object of type *Num* we set the next as the next one in the *num list*
- For each carrier, we set the start location as the deposit and the number of boxes in it as *num0* which corresponds to 0

```
## Next relation  
for x in range(0, len(nums) - 1) :  
    f.write(f"(isNext {nums[x]} {nums[x+1]})\n")  
  
## Carrier state  
for car in carrier :  
    f.write(f"(isCarrierInLocation {car} depot)\n")  
    f.write(f"(carrierHasBoxes {car} {nums[0]})\n")
```

Figure 20: Part 2 - New State Init

As for the goal, it does not change.

2.1.3 Execution with solvers

Once created some problems using the generator, we tried to solve them using the same solver as in part 1, but none of those actually used the carrier and this is because we have not introduced the action cost: once introduced the cost of an action using the carrier is actually more beneficial than going back and forth to the depot. An example of this is shown in figure 21.

```

Searching ('.' = every 50 search steps):
solution found:
first_solution_cpu_time: 0.06

Plan computed:
Time: (ACTION) [action Duration; action Cost]
0.0000: (PICKBOX DRONE1 CRATE4 DEPOT) [D:1.00; C:1.00]
1.0000: (MOVEDRONE DRONE1 DEPOT LOC1) [D:1.00; C:1.00]
2.0000: (DELIVERBOX DRONE1 CRATE4 FOOD PERSON3 LOC1) [D:1.00; C:1.00]
3.0000: (MOVEDRONE DRONE1 LOC1 DEPOT) [D:1.00; C:1.00]
4.0000: (PICKBOX DRONE1 CRATE5 DEPOT) [D:1.00; C:1.00]
5.0000: (MOVEDRONE DRONE1 DEPOT LOC3) [D:1.00; C:1.00]
6.0000: (MOVEDRONE DRONE1 LOC3 LOC1) [D:1.00; C:1.00]
7.0000: (DELIVERBOX DRONE1 CRATE5 MEDICINE PERSON1 LOC1) [D:1.00; C:1.00]
8.0000: (MOVEDRONE DRONE1 LOC1 DEPOT) [D:1.00; C:1.00]
9.0000: (PICKBOX DRONE1 CRATE1 DEPOT) [D:1.00; C:1.00]
10.0000: (MOVEDRONE DRONE1 DEPOT LOC5) [D:1.00; C:1.00]
11.0000: (DELIVERBOX DRONE1 CRATE1 FOOD PERSON4 LOC5) [D:1.00; C:1.00]
12.0000: (MOVEDRONE DRONE1 LOC5 LOC4) [D:1.00; C:1.00]
13.0000: (MOVEDRONE DRONE1 LOC4 DEPOT) [D:1.00; C:1.00]
14.0000: (PICKBOX DRONE1 CRATE2 DEPOT) [D:1.00; C:1.00]
15.0000: (MOVEDRONE DRONE1 DEPOT LOC4) [D:1.00; C:1.00]
16.0000: (DELIVERBOX DRONE1 CRATE2 FOOD PERSON5 LOC4) [D:1.00; C:1.00]
17.0000: (MOVEDRONE DRONE1 LOC4 DEPOT) [D:1.00; C:1.00]
18.0000: (PICKBOX DRONE1 CRATE3 DEPOT) [D:1.00; C:1.00]
19.0000: (MOVEDRONE DRONE1 DEPOT LOC1) [D:1.00; C:1.00]
20.0000: (DELIVERBOX DRONE1 CRATE3 FOOD PERSON1 LOC1) [D:1.00; C:1.00]
21.0000: (MOVEDRONE DRONE1 LOC1 DEPOT) [D:1.00; C:1.00]

Solution number: 1
Total time: 0.06
Search time: 0.04
Actions: 22
Duration: 22.000
Plan quality: 22.000

```

Figure 21: Part 2 - Example No Carrier Used

2.2 Exercise 2

2.2.1 Modify Domain

In order to introduce the action cost in the domain we added what shown in figure 22:

- *total-cost*, is the total cost value that we are going to minimize
- *fly-cost*, is the cost of moving the drone from one location to other

```

(:functions
  (total-cost) 6
  (fly-cost ?from - Location ?to - Location) 2
)

```

Figure 22: Part 2 - Functions

As for the costs of actions, we considered no-fly action with a cost of 1, while we considered the cost of fly actions as the *fly-cost*, as shown in figure ??.

2.2.2 Modify Problem Generator

We have to modify the problem generator as well in order to generate problems considering the distances between locations.



Figure 23: Part 2 - Fly Actions Costs

First of all using the two functions shown in figure 24 we can build the distance metric:

- The distance between two locations is calculated as the euclidean distance between the two locations (any location has its own coordinates in an euclidean space where the depot has coordinates (0,0))
- The cost of a flight is calculated as the distance plus 1

```

# This function returns the euclidean distance between two locations.
# The locations are given via their integer index. You won't have to
# use this other than indirectly through the flight cost function.
def distance(location_coords, location_num1, location_num2):
    x1 = location_coords[location_num1][0]
    y1 = location_coords[location_num1][1]
    x2 = location_coords[location_num2][0]
    y2 = location_coords[location_num2][1]
    return math.sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)

# This function returns the action cost of flying between two
# locations supplied by their integer indexes. You can use this
# function when you extend the problem generator to generate action
# costs.
def flight_cost(location_coords, location_num1, location_num2):
    return int(distance(location_coords, location_num1, location_num2)) + 1

```

Figure 24: Part 2 - Distance Generation

We can then initialize the *flight-cost* function as shown in figure 25: for each pair of location, included a location and itself, the *fly-cost* function is initialized as the value returned by the *fly-cost* function. On the other hand, the *total-cost* is initialized as 0.

Finally we can add a directive to minimize the *total-cost*, as shown in figure 26.

Examples of generated problems are shown in figure 27.

2.2.3 Execution with Solvers

We can divide planners in:

```

## Distance metric
for firstLocIndex in range(0, len(location)) :
    for secLocIndex in range(0, len(location)) :
        cost = flight_cost(location_coors, firstLocIndex, secLocIndex)
        loc_1 = location[firstLocIndex]
        loc_2 = location[secLocIndex]
        f.write(f"= (fly-cost {loc_1} {loc_2}) {cost})\n")
f.write("\n")

## TotalCost init
f.write(f"= (total-cost) 0)\n")

```

Figure 25: Part 2 - Distance Init

```

## Metric
f.write("(:metric minimize (total-cost))\n")

f.write(")\n")

```

Figure 26: Part 2 - Metric Minimization

```

(isNext num0 num1)
(isNext num1 num2)
(isNext num2 num3)
(isNext num3 num4)

(isCarrierInLocation carrier1 depot)
(carrierHasBoxes carrier1 num0)
(= (fly-cost depot depot) 1)
(= (fly-cost depot loc1) 223)
(= (fly-cost depot loc2) 68)
(= (fly-cost depot loc3) 182)
(= (fly-cost depot loc4) 131)
(= (fly-cost loc1 depot) 223)
(= (fly-cost loc1 loc1) 1)
(= (fly-cost loc1 loc2) 189)
(= (fly-cost loc1 loc3) 67)
(= (fly-cost loc1 loc4) 96)
(= (fly-cost loc2 depot) 68)
(= (fly-cost loc2 loc1) 189)
(= (fly-cost loc2 loc2) 1)
(= (fly-cost loc2 loc3) 134)
(= (fly-cost loc2 loc4) 94)
(= (fly-cost loc3 depot) 182)
(= (fly-cost loc3 loc1) 67)
(= (fly-cost loc3 loc2) 134)
(= (fly-cost loc3 loc3) 1)
(= (fly-cost loc3 loc4) 55)
(= (fly-cost loc4 depot) 131)
(= (fly-cost loc4 loc1) 96)
(= (fly-cost loc4 loc2) 94)
(= (fly-cost loc4 loc3) 55)
(= (fly-cost loc4 loc4) 1)

(= (total-cost) 0)

```

```

(= (fly-cost depot depot) 1)
(= (fly-cost depot loc1) 223)
(= (fly-cost depot loc2) 68)
(= (fly-cost depot loc3) 182)
(= (fly-cost depot loc4) 131)
(= (fly-cost loc1 depot) 223)
(= (fly-cost loc1 loc1) 1)
(= (fly-cost loc1 loc2) 189)
(= (fly-cost loc1 loc3) 67)
(= (fly-cost loc1 loc4) 96)
(= (fly-cost loc2 depot) 68)
(= (fly-cost loc2 loc1) 189)
(= (fly-cost loc2 loc2) 1)
(= (fly-cost loc2 loc3) 134)
(= (fly-cost loc2 loc4) 94)
(= (fly-cost loc3 depot) 182)
(= (fly-cost loc3 loc1) 67)
(= (fly-cost loc3 loc2) 134)
(= (fly-cost loc3 loc3) 1)
(= (fly-cost loc3 loc4) 55)
(= (fly-cost loc4 depot) 131)
(= (fly-cost loc4 loc1) 96)
(= (fly-cost loc4 loc2) 94)
(= (fly-cost loc4 loc3) 55)
(= (fly-cost loc4 loc4) 1)

(= (total-cost) 0)

```

```

(= (fly-cost depot loc2) 68)
(= (fly-cost depot loc3) 182)
(= (fly-cost depot loc4) 131)
(= (fly-cost depot loc5) 154)
(= (fly-cost loc1 depot) 223)
(= (fly-cost loc1 loc1) 1)
(= (fly-cost loc1 loc2) 189)
(= (fly-cost loc1 loc3) 67)
(= (fly-cost loc1 loc4) 96)
(= (fly-cost loc1 loc5) 74)
(= (fly-cost loc2 depot) 68)
(= (fly-cost loc2 loc1) 189)
(= (fly-cost loc2 loc2) 1)
(= (fly-cost loc2 loc3) 134)
(= (fly-cost loc2 loc4) 94)
(= (fly-cost loc2 loc5) 115)
(= (fly-cost loc3 depot) 182)
(= (fly-cost loc3 loc1) 67)
(= (fly-cost loc3 loc2) 134)
(= (fly-cost loc3 loc3) 1)
(= (fly-cost loc3 loc4) 55)
(= (fly-cost loc3 loc5) 34)
(= (fly-cost loc4 depot) 131)
(= (fly-cost loc4 loc1) 96)
(= (fly-cost loc4 loc2) 94)
(= (fly-cost loc4 loc3) 55)
(= (fly-cost loc4 loc4) 1)
(= (fly-cost loc4 loc5) 24)
(= (fly-cost loc5 depot) 154)
(= (fly-cost loc5 loc1) 74)
(= (fly-cost loc5 loc2) 115)
(= (fly-cost loc5 loc3) 34)
(= (fly-cost loc5 loc4) 24)
(= (fly-cost loc5 loc5) 1)

(= (total-cost) 0)

```

(a) Three locations

(b) Four locations

(c) Five locations

Figure 27: Part 2 - Example of problems generated

- *Anytime*. An anytime planner is a type of planning system that is capable of providing solutions of varying quality within a specified time frame or computational budget. Unlike optimal planners, which prioritize finding the best solution regardless of the time taken, anytime planners focus on producing usable solutions quickly and improving them over time if additional resources are available.
- *Optimal*. An optimal planner is a type of planning system that aims to find solutions to planning problems that are guaranteed to be the best possible according to a specified optimization criterion. These planners prioritize finding the best solution over providing solutions quickly, often sacrificing runtime efficiency for optimality.
- *Portfolio*. A portfolio planner is a planning system that combines multiple planners or planning algorithms into a single framework to improve overall performance and robustness across different problem instances. Portfolio planners leverage the strengths of individual planners by selecting the most suitable one for each specific planning problem or instance.

Let us now analyze some solvers and their performances:

- *metricff*, which is an optimal planner
- *fastdownword* with aliases
 - *LAMA*, which is an anytime planner
 - *BJOLP*, which is an optimal planner
 - *FDSS2*, a portfolio planner

In this case not to have too much long test time, we did only one execution for each planner for each problem. The maximum dimension solved by each planner is shown in table 4. As we can see in this table:

- With *metricff* we get a small problem dimension because after the dimension of 10 it gave error
- With *LAMA* we reach a biggest problem size in the time limit, but this is because *LAMA* is an anytime planner which returns the first solution, not the best one and, as we can see, the cost is high
- As for *BJOLP* and *FDSS2*, the performances are quite similar but *FDSS2* reaches a slightly bigger problem dimension

Planner	Problem Dimension	Time [s]	Actions	Cost
metricff	10	0.0341	41	3658.0
LAMA	113	53.7885	452	34456.0
BJOLP	6	6.64624	30	581.0
FDSS2	8	3.51162	40	799.0

Table 4: Part 2 - Maximum Solvable Problems

3 Part 3

In this part we are going to parallelise the problem.

3.1 Exercise 1

Actions that can not be performed in parallel:

- Two drones cannot pick up the same box at the same time neither from the depot nor from a carrier
- A drone cannot pickup two different boxes at the same time
- A drone cannot move to two different locations in the same time
- Two drones cannot move the same carrier at the same time
- A drone cannot move a carrier while an other drone is loading it or picking up a box from it
- A drone can not deliver the same box to two different people in the same place

On the other hand two drones can put a box/pick box on/from the carrier at the same time, paying attention at numerical vars changes and checks.

3.2 Exercise 2

3.2.1 Changing the Domain

The only thing we have to modify in the domain is the implementation of all actions.

In figure 28 are shown the actions with no carrier.

- *PickBox* (Figure 28a)
 - Conditions
 - * At Start
 - We check for the drone to be free at the beginning of the action

- We check for the box to be in the same location as the drone; we put this condition as an *start* condition because in the effect we changed it as a *start* effect
 - * Over All
 - The drone has to be in the same location for the whole action time
 - Effects
 - * At Start
 - We mark the box as no longer in the location, so any other drone in the same location can not pick up the same box
 - We mark the drone as no longer free, so the drone can not pick up an other box in the same place at the same time
 - * At End
 - The final effect of the action is that the drone now has the box
- *DeliverBox* (Figure 28b)
 - Conditions
 - * At Start
 - We check that the drone has the box
 - * Over All
 - Person and Drone have to be in the same location for the whole time of the action
 - The Box has to contain the same supply during the action (it actually never changes)
 - Effects
 - * At Starts
 - We mark *droneHasBox* as false, because otherwise the same drone could deliver the same box to two different people in the same location
 - * At End
 - We mark the drone as free at the end because otherwise it could pick up an other box in the meanwhile
 - We mark the *personHas* as true. Putting it as a *At End* effect could let two drones deliver different boxes containing the same supply to the same person in the same moment and, as a consequence, we would lack of a box; nevertheless, if we lack of a box to deliver, that branch of search would not have a solution in the solver, making the solver search for a solution where this condition never verifies.
- *MoveDrone* (Figure 28c)

- Conditions
 - * At Start
 - We have to check that the drone is actually in the start location
- Effects
 - * At Start
 - We mark the drone as no longer in a location, because otherwise the drone could move to two different locations at the same time
 - * At End
 - We mark the drone with the new location

```

(:durative-action PickBox
:parameters (?d - Drone ?b - Box ?l - Location)
:duration (= ?duration 5)
:condition (and
  (at start (and
    (isDroneFree ?d)
    (isBoxInLocation ?b ?l)
  ))
  (over all (and
    (isDroneInLocation ?d ?l)
  ))
)
:effect (and
  (at start (and
    (not (isBoxInLocation ?b ?l))
    (not (isDroneFree ?d))
  ))
  (at end (and
    (droneHasBox ?d ?b)
    (increase (total-cost) 1)
  ))
)
)

```

(a) Part 3 - PickBox

```

(:durative-action DeliverBox
:parameters (?d - Drone ?b - Box ?s - Supply ?p - Person ?l - Location)
:duration (= ?duration 5)
:condition (and
  (at start (and
    (droneHasBox ?d ?b)
  ))
  (over all (and
    (isDroneInLocation ?d ?l)
    (isPersonInLocation ?p ?l)
    (boxContains ?b ?s)
  ))
)
:effect (and
  (at start (and
    (not (droneHasBox ?d ?b))
  ))
  (at end (and
    (isDroneFree ?d)
    (personHasBox ?p ?b)
    (increase (total-cost) 1)
  ))
)
)

```

(b) Part 3 - DeliverBox

```

(:durative-action MoveDrone
:parameters (?d - Drone ?from - Location ?to - Location)
:duration (= ?duration (fly-cost ?from ?to))
:condition (and
  (at start (and
    (isDroneInLocation ?d ?from)
  ))
)
:effect (and
  (at start (and
    (not (isDroneInLocation ?d ?from))
  ))
  (at end (and
    (isDroneInLocation ?d ?to)
    (increase (total-cost) (fly-cost ?from ?to))
  ))
)
)

```

(c) Part 3 - MoveDrone

Figure 28: Durative Actions - No Carrier

In figure 29 are shown actions involving the carrier.

- *PutBoxOnCarrier* (Figure 29a)
 - Conditions

- * At Start
 - The drone has to have the box that it is loading
 - The carrier has to have *startNum* boxes inside
- * Over All
 - Drone and Carrier have to be in the same place for the whole time of the action: this means that no drone can move the carrier while an other drone is putting a box in it
 - The continuity relationship between number has always to be respected (actually it is never changed)
- Effects
 - * At Start
 - We change the number of boxes inside the carrier from *startNum* to *endNum*: making this at the beginning of the action, makes it possible for another drone to put another box inside the carrier without having conflict in the number of boxes inside the carrier
 - * At End
 - The drone no longer has the box and it is now free
 - The box now is in the carrier
- *PickBoxFromCarrier* (Figure 29b)
 - Conditions
 - * At Start
 - The drone has to be free to pick up the box
 - The box has to be in the carrier to be picked up
 - The carrier has to have *startNum* boxes
 - * Over All
 - Drone and carrier have to be in the same location during the whole time; otherwise another drone could move the carrier in the meanwhile.
 - The the continuity relationship between number has always to be respected (actually it is never changed)
 - Effects
 - * At Start
 - The box is no longer in the carrier: no other drone can pick it up in the meantime
 - The drone is no longer free: it cannot pick up an other box in the meantime
 - We change the number of boxes in the carrier from *startNum* to *endNum*: in this way an other drone can pick up a box from the same carrier without having conflict in the number of boxes inside the carrier

- * At End
 - The drone has picked up the box
- *MoveCarrier* (Figure 29c)
 - Conditions
 - * At Start
 - Drone and Carrier have to be in the same location
 - * Over All
 - The Drone has to be free during all the action, because otherwise it could pick up something while moving: it is not concretely free, but it is actually carrying the carrier
 - Effects
 - * At Start
 - We mark drone and carrier location as not the start location: this is because otherwise the drone could move the carrier in two different places at the same time
 - * At End
 - We mark drone and carrier location as the final location

```

(durative-action PutBoxOnCarrier
  parameters (D - Drone ?c - Carrier ?l - Location ?b - Box ?startNum - Num ?endNum - Num)
  duration (= ?duration 5)
  condition (and
    (at start (and
      (droneHasBox ?b ?b)
      (carrierHasBoxes ?c ?startNum)
    ))
    (over all (and
      (isDroneInLocation ?b ?l)
      (isCarrierInLocation ?c ?l)
      (isNext ?startNum ?endNum)
    ))
  )
  effect (and
    (at start (and
      (not (carrierHasBoxes ?c ?startNum))
      (carrierHasBoxes ?c ?endNum)
    ))
    (at end (and
      (not (droneHasBox ?b ?b))
      (isDroneFree ?b)
      (isBoxInCarrier ?c ?b)
      (increase (total-cost) 1)
    ))
  )
)

```

(a) Part 3 - PutBoxOnCarrier

```

(durative-action PickBoxFromCarrier
  parameters (D - Drone ?c - Carrier ?l - Location ?b - Box ?startNum - Num ?endNum - Num)
  duration (= ?duration 5)
  condition (and
    (at start (and
      (isDroneFree ?d)
      (isBoxInCarrier ?c ?b)
      (carrierHasBoxes ?c ?startNum)
    ))
    (over all (and
      (isDroneInLocation ?d ?l)
      (isCarrierInLocation ?c ?l)
      (isNext ?endNum ?startNum)
    ))
  )
  effect (and
    (at start (and
      (not (isBoxInCarrier ?c ?b))
      (not (carrierHasBoxes ?c ?startNum))
      (not (isDroneFree ?d))
      (carrierHasBoxes ?c ?endNum)
    ))
    (at end (and
      (droneHasBox ?d ?b)
      (increase (total-cost) 1)
    ))
  )
)

```

(b) Part 3 - PickBoxFromCarrier

```

(durative-action MoveCarrier
  parameters (D - Drone ?c - Carrier ?from - Location ?to - Location)
  duration (= ?duration (fly-cost ?from ?to))
  condition (and
    (at start (and
      (isDroneInLocation ?d ?from)
      (isCarrierInLocation ?c ?from)
    ))
    (over all (isDroneFree ?d))
  )
  effect (and
    (at start (and
      (not (isDroneInLocation ?d ?from))
      (not (isCarrierInLocation ?c ?from))
    ))
    (at end (and
      (isDroneInLocation ?d ?to)
      (isCarrierInLocation ?c ?to)
      (increase (total-cost) (fly-cost ?from ?to))
    ))
  )
)

```

(c) Part 3 - MoveCarrier

Figure 29: Durative Actions - Carrier

3.2.2 Changing the problem generator

As we removed the metric minimization added in part 2 in the domain definition, we removed the it from the problem as well.

3.2.3 Tests and Performances

We tried the new defined domain using the *Optic* planner. To test the correct generation of a plan we tested it with two drones, two carriers and a size 5 problem; the generated plan is shown in figure 30a. As we can see from the plan two drones are actually used in parallel in a consistent way: for example they do not pick up the same box. In this example we can also see how the carrier is not used: this is probably due to the fact that the optimizer focuses on produce parallel actions while other types of optimization are not well managed. To check that parallel plans are generated using the carrier, we commented the action *MoveDrone*: in this way the solver is forced to use the *MoveCarrier* action in order to move the drone. In figure 30b is shown an example of parallel plan using the carrier.

```

: States evaluated: 135
: Cost: 1047.001
: Time 0.38
0.000: (pickbox drone1 crate5 depot) [5.000]
0.000: (pickbox drone2 crate1 depot) [5.000]
5.000: (movedrone drone1 depot loc1) [223.000]
5.000: (movedrone drone2 depot loc5) [154.000]
159.000: (deliverbox drone2 crate1 food person4 loc5) [5.000]
164.000: (movedrone drone2 loc5 loc4) [24.000]
188.001: (movedrone drone2 loc4 depot) [131.000]
228.000: (deliverbox drone1 crate3 medicine person1 loc1) [5.000]
233.000: (movedrone drone1 loc1 depot) [223.000]
319.001: (pickbox drone2 crate2 depot) [5.000]
324.001: (movedrone drone2 depot loc4) [131.000]
455.001: (deliverbox drone2 crate2 food person5 loc4) [5.000]
456.000: (pickbox drone1 crate3 depot) [5.000]
460.001: (movedrone drone2 loc4 depot) [131.000]
461.000: (movedrone drone1 depot loc1) [223.000]
591.001: (pickbox drone2 crate4 depot) [5.000]
596.001: (movedrone drone2 depot loc1) [223.000]
684.000: (deliverbox drone1 crate3 food person3 loc1) [5.000]
689.000: (movedrone drone1 loc1 depot) [223.000]
619.001: (deliverbox drone2 crate4 food person1 loc1) [5.000]
824.001: (movedrone drone2 loc1 depot) [223.000]

```

```

: Cost: 1087.009
: Time 0.96
0.000: (pickbox drone1 crate5 depot) [5.000]
5.001: (putboxoncarrier drone1 carrier1 depot crate5 num0 num1) [5.000]
10.001: (movecarrier drone2 carrier1 depot loc1) [223.000]
10.002: (pickbox drone1 crate1 depot) [5.000]
15.003: (putboxoncarrier drone1 carrier2 depot crate1 num0 num1) [5.000]
20.003: (movecarrier drone1 carrier2 depot loc5) [154.000]
124.003: (pickboxfromcarrier drone1 carrier2 loc5 crate1 num1 num0) [5.000]
179.004: (deliverbox drone1 crate1 food person4 loc5) [5.000]
184.004: (movecarrier drone1 carrier2 loc5 loc4) [24.000]
208.005: (movecarrier drone1 carrier2 loc4 depot) [131.000]
233.001: (pickboxfromcarrier drone2 carrier1 loc1 crate5 num1 num0) [5.000]
238.002: (deliverbox drone2 crate5 medicine person1 loc1) [5.000]
243.002: (movecarrier drone2 carrier1 loc1 depot) [223.000]
339.005: (pickbox drone1 crate2 depot) [5.000]
344.006: (putboxoncarrier drone1 carrier2 depot crate2 num0 num1) [5.000]
349.006: (movecarrier drone1 carrier2 depot loc4) [131.000]
406.002: (pickbox drone2 crate3 depot) [5.000]
471.003: (putboxoncarrier drone2 carrier1 depot crate3 num0 num1) [5.000]
476.003: (movecarrier drone2 carrier1 depot loc1) [223.000]
480.006: (pickboxfromcarrier drone1 carrier2 loc4 crate2 num1 num0) [5.000]
485.007: (deliverbox drone1 crate2 food person5 loc4) [5.000]
490.007: (movecarrier drone1 carrier2 loc4 depot) [131.000]
621.007: (pickbox drone1 crate4 depot) [5.000]
626.008: (putboxoncarrier drone1 carrier2 depot crate4 num0 num1) [5.000]
631.008: (movecarrier drone1 carrier2 depot loc1) [223.000]
699.003: (pickboxfromcarrier drone2 carrier1 loc1 crate3 num1 num0) [5.000]
704.004: (deliverbox drone2 crate3 food person3 loc1) [5.000]
709.004: (movecarrier drone2 carrier1 loc1 depot) [223.000]
834.009: (pickboxfromcarrier drone1 carrier2 loc1 crate4 num1 num0) [5.000]
839.009: (deliverbox drone1 crate4 food person1 loc1) [5.000]
864.009: (movecarrier drone1 carrier2 loc1 depot) [223.000]

```

(a) Part 3 - Parallelism with No Carrier (b) Part 3 - Parallelism using Carrier

Figure 30: Part 3 - Parallelism Example

As for the performances we tested optic doing:

- two executions for each problem
- using only one drone and one carrier
- considering only the first generated solution (param -N of *Optic*)

The data for the biggest problem that the solver could solve in 60 seconds are shown in table 5 while the time trend is shown in figure 31: in the figure we can see a classical exponential pattern as other planners have also shown.

Once found the biggest problem that *Optic* can solve in 60 seconds using only one drone, let us analyze this problem changing the number of parallelisable units i.e. the drones number; the results are shown in 6. As we can see from the

Planner	Problem Dimension	Actions	Time [s]	Cost
optic-clp	35	165	51.36	13813.025

Table 5: Part 3 - Optic max problem dimension

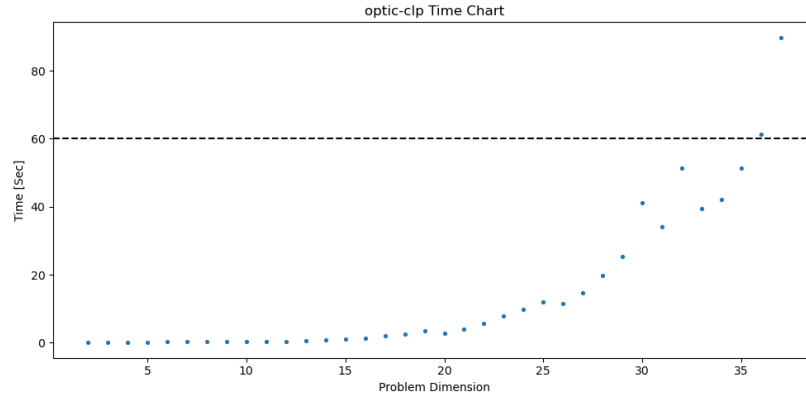


Figure 31: Part 3 - Optic time trend

table, moving from one to two drones makes the number of actions decrease, as well as the total cost and this is because using two drones allows the solver to find a parallel solution to the problem; we can see as well how the performance of the solver gets worse, moving from a search time of 1 minute for one drone, to 1 hour with two drones and getting error for three drones.

Problem Dimension	Number of Drones	Time [s]	Actions	Cost
35	1	52.58	165	13813.025
35	2	3191.14	154	6487.005
35	3	-1	-1	-1

Table 6: Part 3 - Optic performance for increasing drones number

4 Notes

All the tests have been made using automated scripts collecting data and writing the in csv files.