The Fast Downward Planning System

Moldovan Horia-Andrei

November 2019

1 About Fast Downward

Fast Downward is a classical planning system based on heuristic search. Like other well-known planners such as HSP and FF, Fast Downward is a progression planner, searching the space of world states of a planning task in the forward direction. However, unlike other PDDL planning systems, Fast Downward does not use the propositional PDDL representation of a planning task directly. Instead, the input is first translated into an alternative representation called multivalued planning tasks, which makes many of the implicit constraints of a propositional planning task explicit. Exploiting this alternative representation, Fast Downward uses hierarchical decompositions of planning tasks for computing its heuristic function, called the causal graph heuristic, which is very different from traditional HSP-like heuristics based on ignoring negative interactions of operators.

We will now describe the overall architecture of the planner. Fast Downward is a classical planning system based on the ideas of heuristic forward search and hierarchical problem decomposition. It can deal with the full range of propositional PDDL2.2 (Fox Long, 2003; Edelkamp Hoffmann, 2004), i. e., in addition to STRIPS planning, it supports arbitrary formulae in operator preconditions and goal conditions, and it can deal with conditional and universally quantified effects and derived predicates (axioms). The name of the planner derives from two sources: Of course, one of these sources is Hoffmann's very successful FF ("Fast Forward") planner (Hoffmann Nebel, 2001). Like FF, Fast Downward is a heuristic progression planner, i. e., it computes plans by heuristic search in the space of world states reachable from the initial situation. However, compared to FF, Fast Downward uses a very different heuristic evaluation function called the causal graph heuristic. The heuristic evaluator proceeds "downward" in so far as it tries to solve planning tasks in the hierarchical fashion outlined in the introduction. Starting from top-level goals, the algorithm recurses further and further down the causal graph until all remaining subproblems are basic graph search tasks. Similar to FF, the planner has shown excellent performance: The original implementation of the causal graph heuristic, plugged into a standard best-first search algorithm, outperformed the previous champions in that area, FF and LPG (Gerevini, Saetti, Serina, 2003), on the set of STRIPS benchmarks from the first three international planning competitions (Helmert, 2004).

Fast Downward itself followed in the footsteps of FF and LPG by winning the propositional, non-optimizing track of the 4th International Planning Competition at ICAPS 2004 (referred to as IPC4 from now on).Fast Downward solves a planning task in three phases :

• The translation component is responsible for transforming the PDDL2.2 input into a nonbinary form which is more amenable to hierarchical planning approaches. It applies a number of normalizations to compile away syntactic constructs like disjunctions which are not directly supported by the causal graph heuristic and performs grounding of axioms and operators. Most importantly, it uses invariant synthesis methods to find groups of related propositions which can be encoded as a single multi-valued variable. The output of the translation component is a multi-valued planning task.

• The knowledge compilation component generates four kinds of data structures that play a central role during search: Domain transition graphs encode how, and under what conditions, state variables can change their values. The causal graph represents the hierarchical dependencies between the different state variables. The successor generator is an efficient data structure for determining the set of applicable operators in a given state. Finally, the axiom evaluator is an efficient data structure for computing the values of derived variables.

• The search component implements three different search algorithms to do the actual planning. Two of these algorithms make use of heuristic evaluation functions: One is the well-known greedy best-first search algorithm, using the causal graph heuristic. The other is called multiheuristic best-first search, a variant of greedy best-first search that tries to combine several heuristic evaluators in an orthogonal way; in the case of Fast Downward, it uses the causal graph and FF heuristics. The third search algorithm is called focused iterative-broadening search; it is closely related to Ginsberg and Harvey's (1992) iterative broadening. It is not a heuristic search algorithm in the sense that it does not use an explicit heuristic evaluation function. Instead, it uses the information encoded in the causal graph to estimate the "usefulness" of operators towards satisfying the goals of the task.

2 How does Fast Downward work

A rough explanation on how the program works would be the following. The first thing we want to do is to build the planner using the ./build.py command. This will create our default build release in the directory downward/builds. By default, build.py uses all cores for building the planner. More complex builds or custom build could be used as well. To run Fast Downward we have to use the ./fast-downward.py driver script. Besides this there must be at minimum the PDDL input files and the search options specified.

So in other words the planner receives as input two .pddl files, a domain.pddl and a problem.pddl file plus a search algorithm and a heuristic function.

3 Planning and Artificial Intelligence

We could consider the fact that planning problems are a subclass of AI problems where we need to come up with a plan that moves us from an initial state to a goal state. Usually we have:

- A known starting state: information about the world we know right now.
- A set of possible effects: actions which can change the world, through which we want to reach the goal state.
- A goal state: the final state that we wish to reach.

Planning problems are essentially state space search problems, an approach which builds a graph where nodes are states and edges are state transitions (effects), the goal being to go from the starting state (starting node) through effects to an end state which satisfies some predicates. The artificial intelligence consists in the way of reaching the desired end state, by continuously trying different actions from different states and each time choosing the best one according to the given heuristic.

3.1 About PDDL language

The Planning Domain Definition Language(PDDL) is an attempt to standardize Artificial Intelligence (AI) planning languages. It was first developed by Drew McDermott and his colleagues in 1998 (inspired by STRIPS and ADL among others) mainly to make the 1998/2000 International Planning Competition (IPC) possible, and then evolved with each competition.

The model of the planning problem is separated in two major parts: the domain description and its related problem description. Such a separation is useful because this way we can split the needed elements in two categories, the ones which are present in every specific problem of the problem domain (elements contained in the domain description), and those elements which are specific to each planning problem (contained in the problem description). Therefore an useful consequence might be the fact that several problem descriptions could be associated to the same description domain, just like how in OOP several instances of the same class might be created. When using Fast Downward a such pair of domain-description and problem-description is given as input and the output will be a sequence of sequential or even in-parallel actions which describes the solution of the as input given problem domain. So basically the task of the planner is to receive a problem specification given in .pddl formal and using specified functions and heuristics to return an answer or a solution to the problem. Now lets take a look at the contents of a PDDL1.2 domain and problem description in general.

The domain description consisted of a domain-name definition, definition of requirements (to declare those model-elements to the planner which the PDDL-model is actually using), definition of object-type hierarchy (just like a class-hierarchy in OOP), definition of constant objects (which are present in every problem in the domain), definition of predicates (templates for logical facts), and also the definition of possible actions (operator-schemas with parameters,

which should be grounded/instantiated during execution). Actions had parameters (variables that may be instantiated with objects), preconditions and effects. The effects of actions could be also conditional (when-effects).

The problem description consisted of a problem-name definition, the definition of the related domain-name, the definition of all the possible objects (atoms in the logical universe), initial conditions (the initial state of the planning environment, a conjunction of true/false facts), and the definition of goal-states (a logical expression over facts that should be true/false in a goal-state of the planning environment).

Fast Downward supports up to PDDL 2.2, which means that in addition to the above description it supports also numeric variables, plan-metrics, derived predicates (transitivity) and durative actions.

4 About heuristic functions and informed search algorithms

A heuristic technique or a heuristic for short, is any approach to problem solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect or rational, but which is nevertheless sufficient for reaching an immediate, short-term goal. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution. Heuristics can be mental shortcuts that ease the cognitive load of making a decision. In Computer Science a heuristic is a technique designed for solving problems more quickly when classic methods are too slow or they can't even find an exact solution. In this heuristic approach optimality, accuracy or precision is traded for speed. It could be considered as a shortcut.

A heuristic function is a function used in search algorithms in order to help them execute the heuristic behavior described above. It helps them by analyzing the current data at each step and making a decision about the next step to be taken. For example, it may approximate the exact solution.

As mentioned above, Fast Downward takes as input a searching algorithm along with a heuristic function. The better the heuristic function, the better the execution time and the precision of the searching algorithm. This algorithms which work hand in hand with a heuristic function are called informed search algorithms. Unlike uninformed search algorithms, which look through search space for all possible solutions without having any additional information about the search space, the informed ones contain an array of knowledge such as how far from the goal, path cost, how to reach the goal, etc. This knowledge is used in order to help the algorithm explore less to the search space and find more efficiently the goal. Therefore informed search algorithms are more useful for a large searching space. This are graph-based algorithms and how they work is that at each iteration the node with the smallest heuristic is chosen for the next iteration until the solution is found.

Two main algorithms which are compatible with the Fast Downward planner

are the greedy best first search and the A*(astar) algorithms. Let's see how both of this algorithms work in order to fully understand the functionality of the heuristic in their implementation.

First, let us discuss about the Greedy Best First Search algorithm. It is called Best First because it could be seen as a combination between the Depth First Search and Breadth First Search algorithms, because it switches between this two according to the value of the heuristic. Why is it called greedy? Well, it is because at each iteration among all nodes the one with the smallest heuristic is chosen, until the heuristic of the node is zero. The heuristic could be for example an approximation of the distance from each node to the destination. By using this approach we reach our destination by exploring much less of the search space compared to the approach without using the heuristic.

Another important algorithm which uses a heuristic function is A* (astar). This one is very similar to the Dijkstra's shortest path algorithm. But in its implementation we add the heuristic functionality and by doing so a way lesser space is explored until the solution is found. The difference between Dijkstra and A* is the fact that in Dijkstra's implementation we were sorting the nodes in a queue according to their distance from the start node till the reached node, well in A* we sort the nodes by that distance plus the distance chosen by the heuristic. This way we will always be heading in the direction of the goal node, instead of expanding unnecessary nodes.

5 Planning example

We will take at the gripper example in which we will be considering the problem of transporting balls between rooms via a robot. The robot has two grippers and can move between rooms. Each gripper can hold zero or one ball. Our initial state is that everything is in room A and our goal state is that we want to move all the 4 balls in room B.

First let us take a look at the domain file (domain.pddl).

```
(define (domain gripper-strips)
   (:predicates (room ?r)
                (ball ?b)
                (gripper ?g)
                (at-robby ?r)
                (at ?b ?r)
                (free ?g)
                (carry ?o ?g))

   (:action move
       :parameters  (?from ?to)
       :precondition (and  (room ?from) (room ?to) (at-robby ?from))
       :effect (and  (at-robby ?to)
                     (not (at-robby ?from))))



   (:action pick
       :parameters (?obj ?room ?gripper)
       :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
                            (at ?obj ?room) (at-robby ?room) (free ?gripper))
       :effect (and (carry ?obj ?gripper)
                    (not (at ?obj ?room))
                    (not (free ?gripper))))


   (:action drop
       :parameters  (?obj  ?room ?gripper)
       :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
                            (carry ?obj ?gripper) (at-robby ?room))
       :effect (and (at ?obj ?room)
                    (free ?gripper)
                    (not (carry ?obj ?gripper)))))
```

Figure 1: domain.pddl

Here is the place where we describe the predicates (properties of the objects that we are interested in, can be true or false) and the actions (ways of changing the state of the world). We have 7 predicates: (room ?r) - is r a room?; (ball ?b) - is b a ball?; (gripper ?g) - is g a gripper?; (at-robby ?r) - is the robot in room r?; (at ?b ?r) - is the ball b in room r?; (free ?g) - is the gripper free?; (carry ?o ?g) - does gripper g carry an object?

Next we have 3 actions that can be performed by our robot, each function has parameters, preconditions which are checked to be true and an effect which can be seen as the return value of the function, the change it made. The function

move as example takes as parameter two room objects, the one the robot is in and the one the robot wants to reach. In the precondition part it is checked if the rooms given as parameters are actually rooms and if the robot is in the room he is supposed to be (first). Lastly the effect is the fact that the robot is now in the room he has to reach (to) and it is no more in the room he was at first (from). The action pick is the one which makes the robot use one of his gripper to take a ball. When this happens the fact that the ball is located in the room it previously was becomes false. The drop action frees one gripper of the robot and actualizes the new location of the ball with the room given as parameter.

Next, lets take a look at the problem domain.

```
(define (problem strips-gripper4)
   (:domain gripper-strips)
   (:objects rooma roomb ball1 ball2 ball3 ball4 left right)
   (:init (room rooma)
          (room roomb)
          (ball ball1)
          (ball ball2)
          (ball ball3)
          (ball ball4)
          (gripper left)
          (gripper right)
          (at-robby rooma)
          (free left)
          (free right)
          (at ball1 rooma)
          (at ball2 rooma)
          (at ball3 rooma)
          (at ball4 rooma))
   (:goal (and (at ball1 roomb)
               (at ball2 roomb)
               (at ball3 roomb))))
```

Figure 2: problem.pddl

Here will be defined the objects (things in the world that interest us), the initial state (the state of the world that we are in) and the goal state (things that we want to be true). We first declare the objects that interest us, namely two rooms, 4 balls and 2 robot grippers. Next the initial state begins, where we instantiate each of the declared objects as the predicate it is supposed to be. We also instantiate the two grippers to be free, the robot and the balls to be initially placed in room A. Lastly we see the condition that must to be true in order for the goal to be reached.

With the two .pddl files modeled we can now ask Fast Downward to do the planning for us in order to find a solution. In order to do such we must

specify to the planner the location of the problem.pddl file. Besides that we can specify a bunch of options to the planner like translate, validate, etc. The one we are interested for now is the search option. We add it as argument in the command line among with the algorithm and the heuristic function we want to use. In this case we will use A* algorithm with landmark counting heuristics (astar(lmcut()). The planner will create now the space search graph as mentioned above and search for a solution in order to reach the specified goal. When found, the planner will return the solution by mentioning in order the used method and their parameters.

6 Further Work

For my project I will try to extend the functionalities of the example I already explained. I will try to implement more functionalities to the robot and make him to more things, not just moving balls. I'm not sure exactly what I'll do but fact is that I want to be something that has to do with transport. Like transporting some things from a location to a different one. It would be interesting to make use of interactive teamwork in order to achieve a mean of transportation like robots cooperating to move things or to do stuff with them.

7 Bibliographic Study/Research

The exact idea I chose for my project is composed of smaller projects which simulate different problems which have to do with transportation. Firstly I represented the way a taxi would move through the city in order to move people around. After that I wanted to create some type of movement with a bit more constrains, not to let the object move that freely so I moved to the idea of the Hanoi Tower, where the movement of the disk must follow some requirements. So what is exactly the Tower of Hanoi? The Tower of Hanoi (also called the Tower of Brahma or Lucas' Tower and sometimes pluralized as Towers) is a mathematical game or puzzle. It consists of three rods and a number of disks of different sizes, which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.
3. No larger disk may be placed on top of a smaller disk.

The minimal number of moves required to solve a Tower of Hanoi puzzle is 2n 1, where n is the number of disks. With 3 disks, the puzzle can be solved in 7 moves. Besides the theoretical ideas a bit of programming research has been

made in order to properly know how to use the PDDL language. There are a lot of problem definition domain language examples, some of them which even won big prizes so they were a truly helpful way to let me familiarise more with this programming language. When things got complicated, Fast Downward web page was also an useful helping hand. Not to mention the lecture slides we were presented at the Artificial Intelligence course at the university and all the guidance and the answered questions from the laboratory assistants and sometimes some help even from my classmates.

8 Analysis and Design

The actual idea was to make a program which makes use of the artificial intelligence program Fast Downward, by giving him some PDDL files where he will be told to give the most optimal solution to the tower of Hanoi problem. I first started implementing the game with only 3 disks. Then I observed that it can easily find solutions for 10 disks or even more. Thenceforth I started thinking of ways in order to make it more difficult. I thought of adding to the scene two robots which will have the task of moving the disks themselves. The solution improved exponential with the growth of the number of disks. It was working fine but the robots had no interaction between them so I started thinking of a way they could work together in order to achieve some task, still referring to the tower of Hanoi problem. The solution I found was to add more constrains to the robots so I made it that one of the two robots can only work with the left and middle rods and the other one only with the middle and the right rod. So this way if they were for example to move a disk from the left rod to the right rod in the most optimal way, they had to work together in order to achieve the task. In order to make this possible a lot of variables and action had to be declared because the complicated the task gets, the more specifications you have to create in order to create a way to tell the program what you want from him. So you have to properly specify what it has to do and how he has to do it. In order to do that, each individual object has to have his own predicates, which work like instantiating the object, and predicates where he interacts with other objects. The actions are the place where the state of the objects is verified and changed in order to achieve some functionality. Besides the Tower of Hanoi programs I also wanted to do something which is more applicable to the real life, even if it's easier to implement so I decided to implement a taxi simulation, where cars can move persons from a location to another. The hardest part in writing PDDL files are the proper choose of the predicates. If you have already found the way you want to instantiate the objects and where to place them in relation with the other objects then the other tasks become more simple. The same approach works here too. After the role of each object, like the taxi going from a location to another, being initially placed somewhere, the person being in or out of a take and at a specific location, the consistence of the fuel of the car and some others tasks were decided, all that is to be done is to put them together.

9 Implementation

While implementing the programs, the usual model of planning domain definition language was used. So minimum two pddl files must be created: one domain file and at least one problem file. The problem file must contain the used objects, the initial state of the world and the final state which is the goal state we want to reach. The domain file contains the predicates, which are booleans specifying the state of the objects. It also contains actions, which are the only way to change the state of the world.. The actions are composed of parameters, preconditions and effects. They work in the following way: the state of the objects given as parameters is checked to be true, with the help of the predicates, and if none of it is proven to be false, the action happens by changing the state of the objects in the effect component, again by making use of the predicates. So well defined predicates could simplify your work a lot. While implementing the original version of the Tower of Hanoi two kind of objects were defined: Disk and Rod. Predicates were created in order to tell if the disk or the rod have no object on top of them, if one object is on top op another and if one object is smaller than another. The initial state is the one where all the 3 disk are on top on another in ascending way on the left rod and the goal state would be the same stacking of the disks but on the right placed rod. In order to achieve this, the action move was created where we make sure that the rules of Hanoi are not broken and move one disk from a rod to another. Furthermore, 2 robots were added to the implementation of the planning problem with the task of replacing the move action and use their help in order to change the location of the disks. So two new objects have been added: robot one and robot two. The added actions which have to do with the added robots are help us to tell if a disk is carried by a robot, if a robot is carrying a specified disc and if a robot has no disk in hand. The action move was replaced with the actions pick and drop. The same conditions and effects as by the move action were split in two and composed with the new created predicates the two new actions were created. Last modification to the Tower of Hanoi problem was to make the robots cooperate and in order to do that the robots were split into left robot and right robot, because now they are able to place disks only on the left or mid rod, respectively mid or right rod. In order to do this and to still respect the rules of the Hanoi Tower a lot of predicates and actions were needed. The pick and drop actions were replaced with pick right, pick mid, pick left respectively drop right, drop mid, drop left. The take action was also necessary. This one does nothing more than passing a disk from a robot to another. The initial and goal state stays the same in all of the modifications. The last problem I implemented had nothing to do with the Hanoi Tower but is also related to the same field, namely transportation. It does nothing more than moving people from a place to another with a taxi. The objects of this program are the taxis, the people, the locations and the possible status of the fuel. With the help of predicates we set the taxi or the people to a specified location, we specify if a person is in a taxi or not, if the fuel must be filled, and if the locations are connected. The actions are very simple to understand. We

have the get in action which moves a person in a taxi if they are both at the same location, the get out action, which moves the person outside of the taxi and actualises his location, the move action, which moves the location of the taxi from the first to the second location if the taxi has enough fuel and if the locations are connected. Lastly is the fill fuel action, which does nothing more than filling the tank of the taxi if it remains without fuel.

10 Testing and Validation

All the testing and the validation of the programs were doing by repeatedly changing the effect in the problem file and observing if the output coincide with what we said and if the path to the specified output is the correct one. When this was not true, debug was necessarily in order to find where the code breaks. The only way to do this was by finding which actions were not performed or not correctly and taking a look at their preconditions. By doing so, each bug could be fixed with more(4 hours) or less work.

11 External links

Some information in this article can be found on the following pages:

`http://gki.informatik.uni-freiburg.de/papers/helmert-jair06.pdf`
`https://ocharles.org.uk/posts/2018-12-25-fast-downward.html`
`http://www.cs.toronto.edu/~sheila/2542/w09/A1/introtopddl2.pdf`
`https://www.javatpoint.com/ai-informed-search-algorithms`
`https://en.wikipedia.org/wiki/Heuristic_(computer_science)`
`https://www.youtube.com/watch?v=6TsL96NAZCo`