Automated Planning

Planning tool review

Viktoria Biliouri
Maria Ramirez Corrales

January 20, 2021

Contents

1 Introduction

The purpose of this report is to propose a Domain-independent planner, designed under the specifications of the PDDL fragment for IPC 2018. In other words, the planner is implemented so as to be functional for planning problems, composed of the following requirements: STRIPS, action costs, negative preconditions, and conditional effects.

Our work is devided in three main parts: the pddl parser implementation, which is responsible for the extraction of the information included in the pddl files, and the grounding of predicates and operators, the A star algorithm implementation which is used to propose a plan for the solution of the problem, and finally the heuristics implementation. In each of them we will explain you how the code is implemented and the limitations and assumptions we took into consideration. The two first parts correspond to the part of code that we coded together and the last is also divided in two parts, according with the two different heuristics that we developed individually, the Delete Relaxation Heuristic, and the Critical Path Heuristic.

For the implementation of the planning tool, we have chosen C language because of the rarity of pddl planners in C and also because it is the programming language that we are more used to. While coding, we understood that this decision was not favorable of us, as we saw that reading files and storing their contents is more challenging compared with higher level languages, and there is not a possibility of using external libraries for handling pddl files. However, as we had already spent too much time with the project, we decided to finish it in the same language.

2 Parser implementation

The objective of the pddl parser implemented in the 1st step of our project is, to scan all the information provided in the pddl files, to decode and analyse the planning problem, and finally to ground all of its properties (aka grounded atoms, grounded actions)

The planning tool program needs two input arguments to run: first the name of the pddl file that contains the domain of a set of planning problems and then the name of the pddl file that represents one problem. These files correspond to the two files that should be read from the parser and they should be sent as arguments in this same order.

The general logic to search in the input documents is by having a *FILE pointer* for each of the two files (*dom_ptr* for the domain file and *p_ptr* for the problem file), which are moved so as to track the contents of the files. While we are scanning the files with the pointers, the data are stored to a *search* string, used to observe each word and insert it for processing into the corresponding component of the code. Apart from the *search* string, other tracking strings are used also to spot some specific declarations of the pddl code, like for example the *action_finder* used for notifying the definition of an action, etc. Finally, we have a small 3-characters' string (*stop_finder*) that is responsible for understanding the end of a specific part of the pddl code (for instance, the end of a predicate with a ')').

Once we defined the main logic used to read information from the files, we can start with the description. We check if the input files corresponds to a domain and a problem file. Once it is asured, we open both files and we compare the domain name in both files. If it is not the same, the program stops and displays an error message.

Next step is the requirements check. This planning tool will only accept as requirement STRIPS, action costs, negative preconditions and conditional effects, as asked in the specifications. In addition, we added typing requirement, as it is more efficient for the grounding process. If any of the other requirements is found, the program stops and displays an error message.

Then, we implement the type's sorting process. We can notice, according to the grammar of pddl language, that the types can characterize the objects or constants of a planning problem. Apart from this basic functionality, the typing process can become more complex, if the types are also categorized. That is to say, it is possible, in the description of a planning problem, to have a set of types, which belong to the same more general type. In this part of the code, we cover this possibility, by creating an array of typedef structs called *tTypes*, that contain for each general

category of types, the name of this categroy, and an array of strings, where the sub-types' names are stored.

First important data to store is the constants from the domain file and objects from the program file. To this aim, we have created a struct called *Obj*, where we can store the name of the constant or the object and its type according to the typing algorithm. Both the constants and the objects are stored sequentially in a array of object-type structs called *oObj*.

The next data we search for, in the input files are the predicates. In parallel with the predicates' research, we execute their grounding process. In order to handle the grounded predicates we needed to store more information, so we created a struct called *Atom* with multiple fields to store the necessary information. This struct contains the name of the atom, the number of arguments and the arguments. First, we find the name and the number of the arguments of each atom and then, we ground them, to finally store the grounded atoms in an Atoms' array called *aAtom*.

Before the grounding of predicates, we should understand the real size of memory that we will need for the grounded predicates. For this reason, the process of storing the atoms, consists of three repetitive processes, with nested for-loops, so that we can dynamically allocate as much memory as it is needed. Next, we proceed with the measurement of arguments of its predicates, according to which, we designed the combinations for the grounding process. Afterwards, depending on the number of the arguments of each predicate, and depending on the type of each argument, we set a combination of objects/constants, as arguments of the grounded predicates.

In more detail, we construct the same number of nested for-loops as the number of the arguments of each predicate, and we create a combination of objects/constants whose type corresponds to the type of the corresponding argument. Finally for each combination obtained, a new atom is created, and it is added to the array of Atoms.

The last data to take from the domain file the operators. As for the previous data, we have created a new struct called *Action* with a field for the name of the action and four for the predicates represesnting the positive preconditions, the negative preconditions, the positive effects and the negative effects. Finally, we have a field to store the cost of the action. All the grounded actions will be stored in a list of actions called *aAction*.

As it happened with the predicates, the operators should be grounded to create the actions. The logic to ground them is:

- We take the parameters and their types and we store them in a list of objects called *question*.

- We search for the number of the position of the first object of the searched type in *oObj* list and we store this value in an array *inf[]*.
- We search for the number of the position of the last object of the searched type in *oObj* list and we store this value in an array *sup[]*.
- Once this is done for all the parameters, we enter in to a for-loop including multiple if-conditions (depending on the number of parameters) and for loops whose limits are determined by the values stored in *inf[]* and *sup[]* arrays.
- The loops pass through all the possible objects in the interval between inf[i] and sup[i] and it stores those combinations in a list of strings called *combi[]*, except for the combinations that include two or more same objects/constants.
- Then, we search for the preconditions. If we find a *not*, we store them in the negative preconditions field and if not, in the preconditions field. Afterwards, we change their arguments by the ones stored in the *combi* list. We do exactly the same with the effects in their respective fields, and also with the conditional effects, by having different fields for the condition's predicate and the effect's predicate produced under the restrictions of the condition. The recognision of the conditional effect happens using an boolean called *cond_indicator*, that gets the value true (1) if the word *when* is found.
- Finally, when our file pointer meets the increase cost function, the cost of the current action is stored in the concerning struct field. The indication of reaching the cost function, means the end of an action, so our algorithm continues iterating, if there exists another action, or it quits, having all the grounded operators stored.

The number of parameters that an action can have to be grounded in this planning tool is 6. We cannot assure that it works even for 6 because our computers were not powerful enough to allocate the corresponding memory. However, since our parser was designed in accordance to the specifications, we can assume that it can be functional for the planning problems of the IPC, if the capacity of the computation machine permits it.

Here is the end of the parser for the domain file. Then, we search for the initial and the goal states in the problem file. They will be the first two states of the states' list called *states_list*, where all the states created while the execution of the search algorithm will be stored, process which is analyzed in the next chapter.

3 Search algorthm implementation

First of all, we will speak about how we store the states of the system. As previously said, they are stored in a list. We consider that each state should include more information than the atoms included in it. So, we have created a struct with multiple fields:

- state_num: it is the index of the state. It will be useful for storing the states in a simple way.
- anum: it is the number of atoms that a state contains.
- curr_state: it is the list of atoms contained in the state.
- possible_actions: it is the list of possible actions that can be done from this state.
- num_possible_actions: it is the number of possible actions that can be executed from this state and so, the number of next states that the state will have.
- next_states: it is the list of the next states created from the possible actions and stored in the same order with the action which leads to them.
- pred: it is the predecessor of the state.
- f: it is the f value assigned in the A* algorithm, initialized to infinity (a particularly high number for our program), for the next states and 0 for the initial state.
- h: it is the value assigned by the heuristics algorithm.

The chosen search algorithm is the forward searching algorithm called A*. This is the Dijkstra's algorithm taking into account some heuristics values to insert the states in the *nexts* list. So, as in the begining we created this function with all the h=0, this is Dijkstra's algorithm at this point of the report.

In the beginning, we add the initial state in the first position of *nexts* list. We have decided that this list will be an array of integers that stores the *state_num* of the states that will be analysed in the future. The state with the smallest f+h will be always placed in the first position. In parallel, we have another array of integers that stores the value of f+h of the stored states in nexts in the same position with the corresponding state (called *f_in_nexts*). We initialize each place in this array to 10000 as we have tested the planning tool just for small problems and this number is enough to represent the infinite value that the A* algorithm requires. If we tried this planning tool with bigger problems, this value would be changed to a higher one or memory could be reallocated dynamically to be more efficient.

Then, we start a loop with the algorithm. We should assure that the first state in *nexts* list is not the goal (if it is, the loop will finish). We cannot see it in the beginning of the loop, so we do it at the end of the loop, when we have already computed this value for the next round. First action in the loop is erase the first state in *nexts* and, so, in *f_in_nexts*. Then, we compute the next states of the analyzed state. To that aim, we use three functions:

- *search_possible_actions*: this function will search for the possible actions by comparing the atoms in *curr_state*, the preconditions and the negative preconditions of all the grounded actions. At the end of this function, it will fill the *possible_actions* and *num_possible_actions* fields of the state.
- *create_next_states*: it fills the *next_states* field of the analysed state. First, it will copy the atoms present in *curr_state* in all the *next_states*, then it will compare the effect of the actions in *possible_actions* with those atoms and it will copy those who are not present. Finally, it does the same comparison with the negative effects of the same actions and it erases the ones that were in the corresponding *next_states*.
- *copy_next_states_if_new*: this function will make the real link between one state and its sucessor. First, it will check if the state in *next_states[i]* is new in the *states_list*. If it is, it creates a new state in *states_list[index]* and it initialise all the fields. Then it copies the *next_states[i]* in its *curr_state* field and it links the new state to the *next_states*. Finally, if the state in *next_states[i]* already existed in the list, it copies the state in *next_states*.

Depending in the selected heuristics, it executes the heuristics function (explained in the next section). Once it has the h value of each state, it compares the f of each *next_states[i]* with the addition of the f of the current state and the cost of executing the action to lead to this state (stored in *possible_actions*). If the first value is greater than the second, it copies it in the f field of the corresponding *next_states[i]* and adds the current state to the *pred* field of *next_states[i]*. Last task is to add the *next_states[i]* to the *nexts* list in the correct position. It does it thanks to a for-loop where it searches the first value of *f_in_nexts* greater than the addition of the f and the h of the state to include. If the state is previously added in this position, we update the new value of f+h, if not, we copy the state in *nexts* and the value of f+h in the same position in *f_in_nexts*.

To see if the loop has to start again, we add a 0-flag that will be actualised to 1 if *nexts* is not empty or if the first state in *nexts* is the goal.

At the end of this loop, we have all the predecesors of each state and we can now return a plan. This will be done in another loop and stored in an array of integers

called *pi[]*. To search from where it should start the path, we search for the state with the highest index in the *states_list* that is goal state and we add it in the first position of *pi[]*. Then, it goes to its predecessor and continues the same process until it achieves *states_list[0]* and it displays the whole plan of passed states. To make it easier to understand, it prints too the whole *states_list*.

4 Heuristics implementation

In terms of this project, we have implemented two heuristics' computation methods, to produce the heuristics, used in the Astar algorithm. The first computed heuristics are the Delete relaxation heuristics, and the second, the Critical Path heuristics. Since we integrated both our heuristic methods in the full planner's code, we decided to add a little interaction with the user, who is called to choose with which heuristics' method wants the planner to function. So after the user inserts the decision, the responsible for each heuristics function is called, and the planning is done.

4.1 Delete relaxation heuristics

The first heuristic calculation method that is implemented in this project, is implemented by Viktoria and, it is the computation of the delete relaxation heuristic. The main idea of these heuristics is, that for each state in the list of next possible states, we should calculate the minimum cost of each atom in this current state to reach an atom that is contained in the goal state, so that the maximum value of all the costs of reaching all atoms of the goal state from the state examined, represent its heuristic.

First of all, the Delete Relaxation Heuristics, depend on the relaxed problem and not on the real one. That is to say, each relaxed action, used in this process should not produce any negative effects, and the new states after the execution of each compatible action, should have a greater or equal number of predicates than before, as only new predicates are added to the produced states. So, the first step was to produce the relaxed actions, which are the same, with the real problem's actions, but without the negative effects. For this reason we created the function *relaxation*, that creates a new list of actions *relaxed_actions*, which is used as input in the heuristics calculation function, analyzed below. The way that this function works, is that it takes as input the relaxed problem's actions and real problem's actions, it copies each one of the components of the grounded actions (preconditions, negative preconditions, effects) in the relaxed actions, but without copying the negative effects.

Secondly, the calculation of these heuristics is a repetitive process, of finding a solution to a new planning problem, where we define each time as initial state the state we examine, and as goal state, each atom of the real goal state of our planning problem. The planning algorithm used to compute the heuristics, is the Dijkstra'sforward planning algorithm , which works in the same way with A* but, without using the heuristics when updating the next states list, but only using the f values. Thus, we implemented the Dijkstra's algorithm in the Delete Relaxation Heuristics devoted function, called dr_heuristic, with inputs, the state whose heuristic we want to find, the relaxed actions' list, one goal atom, and the number of the relaxed actions.

After we call this function with the above inputs, we search the possible relaxed actions that is possible to happen from each state (using *search_possible_actions* analyzed in the previous chapter), and then we create the next states that can occur (using *create_next_states* and *copy_next_states_if_new* functions analyzed in the previous chapter). Next, our program implements the Dijkstra's algorithm to this state, by updating the nexts' list, each time according to the minimum value of f+c(o) of each next possible state, and by examining repetitively the first state of this list, until we find the specific goal atom inserted as input.

The function we created to check if an atom is also in the goal state is called *is_it_goal_atom*, and it is based in the *is_it_goal* function, created for the research of the goal state from the a* algorithm. In fact instead of discovering if a state includes all the atoms, this function returns the true value (1) if the state we check, includes each time the goal atom that we are interested in, by passing the state and this goal atom as arguments of the function.

When we find this goal atom, the function returns the total f value from the path that the algorithm followed to reach the goal, otherwise, if the nexts' list gets empty and we dont find the goal atom, it returns infinity (actually the number 10000 which works as the infinity to the dimensions of our planner). After the execution of the delete relaxation heuristics function for each goal atom, in the goal state, we end up with one heuristic value for reaching each goal atom. The maximum value of these heuristics, is the one that we store as h in the state. By repeating the process for each next possible state in the a* procedure, we will get one Delete Relaxation Heuristic for each state, so that the a* can take it into consideration for the additions of states in the nexts list and it will work as desired.

4.2 Critical path heuristics

The second type of heuristics, implemented by Maria, is the critical path heuristics. We have use a critical path h^2 heuristics, which means that if a state has more than two atoms, it will be splited into all the possible combinations of two of its atoms. This same action will be done in the goal state if it satisfies the same condition.

To compute the h value of an state that has being depleted, we take the maximal value of h among all the 2-combination new states created to reach the 2-atoms state. The value of each path will be the minimum cost of achieving the 2-atoms goal from it. We consider that the cost of each action is 1, so the h value will be the number of steps done to achieve the goal. This path will be computed in the relaxed problem, as in the previous heuristic.

The loop on created to do so sort depending on the number of states in the goal

state. If it is one or two, there will be only one possible path, so the value it returns is directly assigned to the heuristic field of the state. If it has more than two, we will send the different combinations of two goal atoms and the result will be stored in an array (*heuristic2[][]*). When the computation of all the combinations has finished, we store the maximal value in the h field of the state.

We have created the function *new_cr_heuristic* that will return the minimal cost for state s to reach two goal atoms (h_{min}), therefore, in the A* algorithm loop, after having called *search_possible_actions*, *create_next_states* and *copy_next_states_if_new*, we create a loop that stores all the returned values for the different split 2-atoms goals and takes only the greater one. When the function with all the combinations of 2 goal-atoms is finished, we assign this value to the f field of the state.

Function *new_cr_heuristic* has as arguments the state to whom we want to assign an h value, the list of relaxed operators, the two goal atoms and the number of grounded actions. If the goal state has only one atom, we have to send only one goal atom, so we cannot use this function and we have created another one (*new_cr_heuristic_1goal*) that makes the whole computation as *new_cr_heuristic* but when it compares the analised state with the goal state at the end of the function, it uses the function *is_it_goal_atom* instead of the *is_it_goal_2atom* (a further explanation will be done in next paragraphs).

For these heuristic's functions, we will create two states list in each: *temp_list* and *temp_cr_list*. The first one will store the created states and the second one will store the depleted states. As for the depleted states we need less data, we have created another struct called *State_cr* that stores only the index of the state and two atoms (stored in *curr_state1* and *curr_state2* fields). They will be guided one with *index1* and *index* and the other one with *index_cr* and *index_cr1*. The loop will not analyse twice the same state, it will always create new states in both lists, that is why we need those two bounds.

First of all, this function will copy in the first position of the *temp_list* the initial state, which is the received state. Then, it will enter in the main loop. The first condition is if the number of atoms in the studied state is 1 or more than 1. For the first case, we do not need to depleat the state and we have to start the search with one state. For the case of 2 neither, but we will have two atoms, so, the process will be the same as for the case of 3 or more once we have depleted the state.

For the case of greater than 2, we will need to compute all the combinations of r elements among n elements. We will do so with numbers (for example, if we have n=3, we will store the combinations 01, 02, 03,12,13 and 23) that will be added in the creation of the new *State_cr*.

To this aim, we have done a while-loop that finishes when we achive the number of combinations of n elements taken 2 by 2 multiplied by 2 (the loop should finish when we have filled the list with all the combinations and as each combination has 2 numbers, the list should have the number of combinations multiplied by 2). To compute this number, we created a function (*num_combinations*) that needs to compute the factorial of n, n-2 and 2, so, it calls to another function called *factorial*. The result of the loop, is stored in the list *list_fin*. *n* is the number of atoms in the studied state.

Thanks to the numbers in *list_fin*, we will be able to create the next *States_cr*. We make a loop steping by 2 values and it creates in each tour the *curr_state1* with the position *list_fin[j]* and *curr_state2* with the position *list_fin[j+1]*.

Once those *States_cr* are created we search for the possible actions from them. We cannot use the *search_possible_actions* because those states are not the same as the ones to which this function is made to.

First, we take a look in the negative preconditions of the actions with the same logic as we did in the function *search_possible_actions*. If they are not in the *States_cr*, we can continue in this action. If not, we pass to the next action in the *relaxed_actions* list. Then, if there was not any negative precondition in the *State_cr*, we search for the positive preconditions, and to do so, we have to create different cases loops: one if the number of preconditions is greater than two (that makes impossible the action, so we pass to the next action in the *relaxed_actions* list), another if it has 2, another if it has 1 and a last one if it has no preconditions.

In the case of 2 preconditions, we see if both the *curr_state1* and *curr_state2* are those two preconditions, in the case of 1 precondition we see if one among the two *curr_state* is the precondition and in the case of 0 preconditions, the action is always possible (as we have already passed the filter of the negative precontidions). We add those possible actions to the studied *State*, as those *State_cr* are not the ones to compare with the goal atoms and the next states to be analysed should be created from *temp_list*'s *States*.

The last functionality of the heuristic function is to create the new *State* in the *temp_list* as a result of the possible actions from each *State_cr*. This can be done with the *create_next_states* and *copy_next_state_if_new* because, as we said previously, we added the possible actions in the state. If after having finished those functions, the index it is the same, it means that no new states has been created and so, we cannot continue the path from here and we add value 0 to the *State.h*.

If the index have been incremented (we see it as the result of *copy_next_states_if_new*), we enter in a loop from *index* to *index1* comparing the created states with the goal

atoms (we have created a function *is_it_goal_2atoms* for it based in the results of *is_it_goal_atom* explained in the previous heuristics). This is the point where function *new_cr_heuristic* and *new_cr_heuristic_1goal* are different. In the second one, we use the function *is_it_goal_atom* instead of *is_it_goal_2atoms*.

If it is/they are the goal, the function ends and returns the value of h of the previous state (the analised one). If it is not, we add one to the value of h from the previous state (as we are computing the number of steps to achieve the goal as heuristic) to the next state being analised (the one that was not the goal) and we continue the loop to the next state in the *temp_list*.

The whole function is an increasing for-loop, as the second condition on the for-loop is the index of the list. If the index is not incremented at the end of the tour this means that no new states has been created and so, that there is no path to the goal states. In this case, the function will return the value of 10000 as minimal cost. For bigger problems, this value should be changed to a higher one, but for the problems we tried, it is enough to avoid the algorithm to take this path.

5 Conclusion

To conclude, we would like to mention, that we did our best and implemented an independent domain planner. Although its appropriate functionality for our input pddl problems, we are having expectations to add more functionalities to our planner, in the future, and to make it more generalized for any pddl syntax/fragment, with any requirements, and with problems of any size.