

**15-150 Fall 2018**  
**Project 01 - Documentation**

Zaryab Shahzaib – zshahzai

Sunday 9<sup>th</sup> December, 2018

**Domain Independent Planner - SML**

# 1 Introduction to Planning

Planning is the task of coming up with a sequence of actions that will achieve a goal. The fundamental idea in planning is that given some description of an initial state or states; a goal state or states; and a set of possible actions that an agent can take in an environment, one should be able to find the sequence of actions that take one from the starting state to the goal state.

To solidify the concept of planning and discuss the motivation behind planning, let's consider a very basic scenario where we can use planning to plan our course of action. The problem is as follows:

You are at home and you want to cook your favorite meal, However, you check the refrigerator and you find that you don't have the required ingredients. As a result, you decide to go to the super market to get the required ingredients. This could be considered as a planning problem, in which your initial state is that you are at home and you don't have the ingredients, the final state is that you are at home and you have the ingredients. What would be the appropriate course of action in this scenario to get the ingredients? The appropriate actions in this scenario would be to go out of your home, go to the super market, go to the appropriate sections in the supermarket and get the required ingredients, and then come back home. Now that we've planned a course of action, try to think about your approach towards planning it. You probably simulated the entire scenario in your head, for example you would have thought of going outside the house, then sitting in your car and going to the super-market etc. All of these were intermediary states that were reached due to some action you took in the previous state. As illustrated in Figure 1, the journey from the initial state to the goal state involves intermediary states ( $A_1, A_2, A_j, A_k, A_m$ ), which are achieved by applying the applicable actions to each successive state.

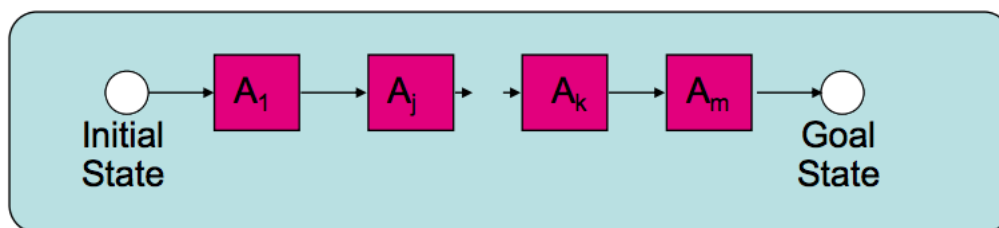


Figure 1: The Planning Process [Digital Image](n.d.)

A more complex example of planning would be the Resource Allocation (scheduling) problem, in which a number of objects need to be processed by machines, which can do various actions, such as polishing, drilling, punching holes, painting, etc. You want that all objects are fully processed at the end, but they start at different states (some are polished but not painted, some have holes but are not polished, etc). How can you allocate each object to each machine so that they are all ready? This planning problem is important to solve as it almost occurs in every domain which requires scheduling or allocation of resources, For example manufacturing industries. The goal is not that one should be able to solve this problem mentally, rather the goal is to find a systematic approach through which a computer can find solutions to such problems.

There can be multiple motivations for planning, for example, to achieve some goal such as moving

an object from one location to another, to reason about the efficiency of a possible course of action that will reach the goal state, or to plan a discourse that will cause another agent to do something. In modern day intelligent systems, planning is a key ability. It has been an area of research in artificial intelligence for over three decades. Planning techniques have been applied in a variety of tasks including robotics, process planning, web-based information gathering, autonomous agents and spacecraft mission control.

## 2 Solving the Planning Problem

### 2.1 Decomposing the Problem

The first step towards solving a planning problem is abstracting the details, so the problem can be modeled by describing an initial state, a goal state, and defining what are the valid steps. If we can solve the abstract version of the problem, we can effectively solve any planning problem. The basic representation language of classical planners is defined as follows:

**Representation of States:** Each state is represented as a conjunction of fluents that are ground, function-less atoms. For example,  $\text{HasBananas} \wedge \text{AtExit}$  might represent the state of a monkey who has bananas and is at the exit door of a cage. A state in the blocks world problem might be  $\text{On}(a,b)$  representing that block a is currently on top of block b. These conjunction of fluents can either be manipulated with logical inference or with set operations. For example, the operation of checking if the goal state has been reached could be done by checking if the goal state is a subset of the current state.

**Representation of Actions:** An action is specified in terms of the preconditions that must hold before it can be executed, and the consequent effects that occur when it is executed (post-conditions). For example, an action for going from location X to location Y is:

**Pre-conditions:**  $\text{at}(x)$

**goXY:**  $\text{at}(x) \rightarrow \text{at}(y)$

**Post-conditions:**  $\neg \text{at}(x) \wedge \text{at}(y)$

An action consists of three main components:

**Name of the action:** Which in this case is goXY

**Pre-Conditions:** Which in this case is  $\text{at}(x)$

**Post-Conditions:** Which in this case is  $\neg \text{at}(x) \wedge \text{at}(y)$

An important thing to note is that the variables x and y could be instantiated to different constants, hence giving birth to different actions.

### 2.2 A Domain Independent Solution

A solution to the planning problem could be described as a state-space search, starting from the initial state, through the intermediary states in the state-space, until the goal state is found. This search method is known as Forward Chaining.

The state space could then be modeled as a directed graph, in which nodes are states and edges between nodes are actions. Note that we won't build the graph as a separate data structure, rather an implicit graph is used to visualize how a search algorithm would traverse the search space. Once we have modeled the state-space as a directed graph, the problem of finding the solution to the planning problem, is transformed into the problem of searching the state-space using graph search Algorithms. The remainder of this section would give an overview of Graph Search Algorithms (GSA), discuss two graph search algorithms, namely Depth-First Search and Breadth-First Search, and put it all together using a planning example.

**An Introduction to Graph Search Algorithms:** The following four main points are the key characteristics of GSA:

- They specify an order to search through the nodes of a graph
- We always start at the **source node** and keep searching until we find the **target node**
- The **frontier** contains nodes that we've seen but haven't explored yet
- In each successive iteration, we take a node off the frontier, and add its neighbors to the frontier

**A General Graph Search Algorithm:**

---

**Algorithm 1** General Graph Search - (Di Caro, 2018)

---

```

1: function GRAPH SEARCH(problem)
2:   set of frontier nodes containing the initial state of problem
3:   loop
4:     if there are no frontier nodes then return No Solution
5:     else choose a frontier node and add it to the visited set
6:     if the node is the goal state then return the path followed to reach the node
7:     else expand the node and add the resulting nodes to the set of frontier nodes, only if
       not in the frontier or visited set

```

---

**Depth-First Search:** The algorithm starts at the root node and explores as far as possible along each branch before backtracking. Depth-First Search can be implemented using a stack for the frontier (LIFO).

**Breadth-First Search:** The algorithm starts at the root node and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. Breadth-First Search can be implemented using a FIFO queue for the frontier.

## 2.3 Putting It All Together:

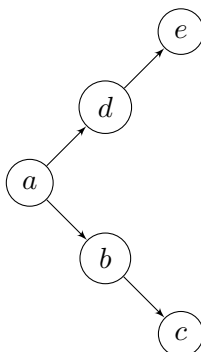
Lets take a simple planning example and simulate how our graph search algorithm would find a solution. The initial state, goal state, and actions are defined as follows:

**init:** {a}

**goal:** {e}

**actions:** {movAB: a → b, movBC: b → c, movAD: a → d, movDE: d → e}

We can model this problem with the following graph, where each successive node is found by applying the respective applicable rules to its parent:



Using DFS or BFS to find the goal state, our frontier would go through the following transitions until we find the goal state, each state of the frontier is labeled with the iteration number.

Iteration	Frontier(DFS)	Frontier(BFS)
1	[a]	[a]
2	[b,d]	[b,d]
3	[c,d]	[c,d]
4	[d]	[c,e]
5	[e]	[e]

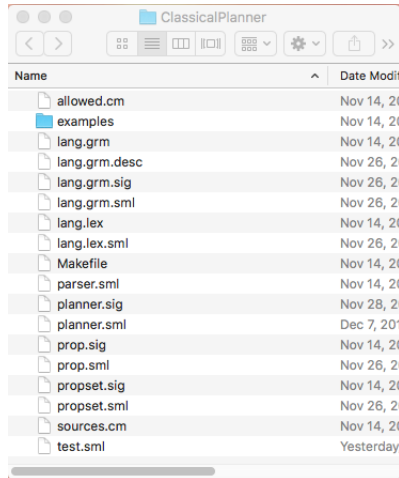
### 3 Usage instructions

The program requires a system with SML/NJ installed. If you do not have SML installed, or have a different version, head to the [SML/NJ Website](#) to install SML/NJ.

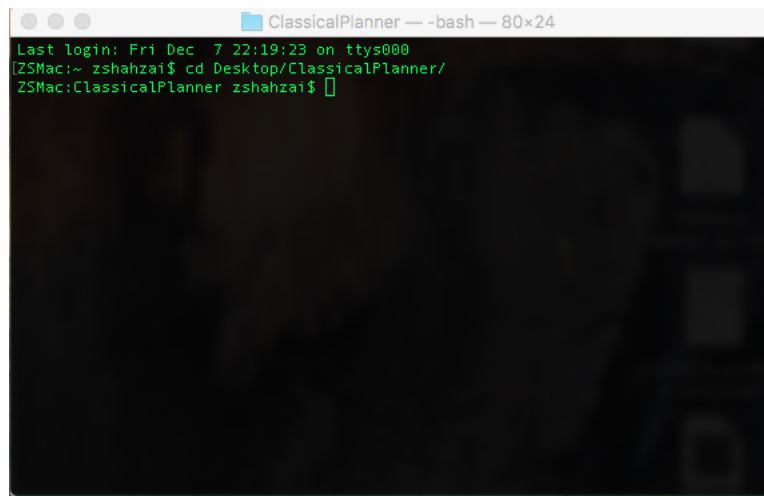
Once you have installed SML/NJ, follow through each of the provided parts to successfully use the program.

#### 3.1 Setting up the Environment

1. Unzip the **ClassicalPlanner.zip** package that was downloaded along with this documentation.



2. Open the Terminal/Command Prompt and navigate to the unzipped folder



3. Once in the unzipped folder, type: `sml -m sources.cm`
4. If you see "[New bindings added.]" at the end of the output, that means that the program has compiled successfully and you can proceed to the next part of the Instructions. If you do not see the message, or you encounter an error, it means that there is something wrong with the file you downloaded. Please download the .zip file again and retry.

```
ClassicalPlanner — run.x86-darwin @SMLcmdname=/usr/local/smlnj/bin/sml @S
[library $SMLNJ-MLRISC/Control.cm is stable]
[library $controls-lib.cm(=$SMLNJ-LIB/Controls)/controls-lib.cm is stable]
[library $smlnj/smlnj-lib/controls-lib.cm is stable]
[loading (sources.cm):prop.sig]
[loading (sources.cm):propset.sig]
[loading (sources.cm):propset.sml]
[loading (sources.cm):lang.grm.sig]
[loading (sources.cm):prop.sml]
[loading (sources.cm):planner.sig]
[loading (sources.cm):planner.sml]
[loading (sources.cm):lang.lex.sml]
[loading (sources.cm):lang.grm.sml]
[loading (sources.cm):parser.sml]
[compiling (sources.cm):test.sml]
[code: 1388, data: 76, env: 601 bytes]
[library $smlnj/viscomp/parser.cm is stable]
[library $smlnj/MLRISC/MLRISC.cm is stable]
[library $SMLNJ-MLRISC/MLRISC.cm is stable]
[library $Graphs.cm(=$SMLNJ-MLRISC)/Graphs.cm is stable]
[library $smlnj/viscomp/execute.cm is stable]
[library $smlnj/internal/smlnj-version.cm is stable]
[library $smlnj/viscomp/x86.cm is stable]
[New bindings added.]
-
```

### 3.2 Constructing the Input

The program takes input problems in a specific format. In order to use the program, you need to follow this specific format so the parser can successfully parse the input and supply it to the program. The input format is as follows:

```
1 | %Comments begin with the % sign.
2 |
3 | RuleName(Vars): PreConditions(Vars) -> PostConditions(Vars).
4 |
5 | #init: {Set containing the initial State}.
6 |
7 | #goal: {Set containing the Goal State}.
```

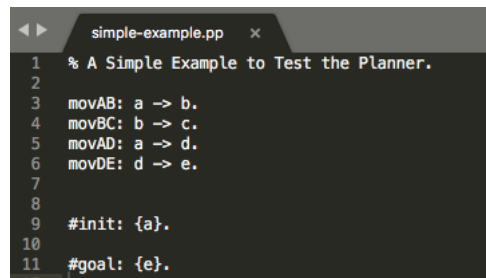
For example, to provide the input for the planning example discussed in **section 2.3**, we would provide input as:

```
1 | % A Simple Example to Test the Planner.
2 |
3 | movAB: a -> b.
4 | movBC: b -> c.
5 | movAD: a -> d.
6 | movDE: d -> e.
7 |
8 | #init: {a}.
9 |
10 | #goal: {e}.
```

If our problem has variables in it, then we would provide input as:

```
1 | %The simplest possible example with variables.
2 |
3 | rule(X): a(X) -> b(X).
4 |
5 | #init: {a(1)}.
6 | #goal: {b(1)}.
```

Once you have made the input for the program, save the file with a **".pp"** extension, in the same folder as the program. For the purpose of demonstration in the next part, we will use the example in section 2.3, and the input format for it described earlier.



```

1  % A Simple Example to Test the Planner.
2
3  movAB: a -> b.
4  movBC: b -> c.
5  movAD: a -> d.
6  movDE: d -> e.
7
8
9  #init: {a}.
10
11 #goal: {e}.

```

### 3.3 Interacting with the Program

After constructing the input for your program, follow the following instructions to interact with the program:

1. Open test.sml

2. Create a new value as:

**val problemX = Parser.parse ("filename.pp")**

problemX can be replaced with any variable name, However filename should be replaced with the name of the input file you just created



```

1  structure Test =
2  struct
3
4      fun planListToString [] = "{}\n"
5        | planListToString (h::t) = "{ " ^
6          (List.foldl (fn (p, acc) => acc ^ "\n " ^ (Planner.toString p) ) (Planner.toString h) t) ^ " }\n"
7
8      (***** TESTS *****)
9
10     val problem0 = Parser.parse ("examples/boat.pp")
11     val plan1 = Planner.planAllBfs problem0
12
13     val problemX = Parser.parse ("simple-example.pp")
14
15
16
17
18 end

```

3. Create a new value as:

**val planX = Planner.functionName (problemX)**

planX can be replaced with any variable name, However problemX should be replaced with the variable name you kept in step 2. Replace functionName with the name of one of the four plan functions: planDfs, planBfs, planAllDfs, or planAllBfs.



```

1 structure Test =
2 struct
3
4   fun planListToString [] = "{}\n"
5     | planListToString (h::t) = "{ " ^
6       (List.foldl (fn (p, acc) => acc ^ "\n " ^ (Planner.toString p) ) (Planner.toString h) t) ^ " }\n"
7
8   (***** TESTS *****)
9
10  val problem0 = Parser.parse ("examples/boat.pp")
11  val plan1 = Planner.planAllBfs problem0
12
13  val problemX = Parser.parse ("simple-example.pp")
14
15  val planX = Planner.planDfs (problemX)
16
17
18
19
20 end

```

4. Save the changes to the file and open a new window in the Terminal/Command Prompt
5. Navigate to the folder which contains the program
6. Type the following command: `sml -m sources.sml`
7. Once the loading completes, and you see "[New bindings added.]", proceed to the next step
8. Type the following command to access the solution to your problem:

**Test.planX**

planX should be replaced with the variable name you kept in step 3

```

ClassicalPlanner — run.x86-darwin @SMLcmdname=/usr/local/smlnj/bin/sml @S...
[library $smlnj/smlnj-lib/controls-lib.cm is stable]
[loading (sources.cm):prop.sig]
[loading (sources.cm):propset.sig]
[loading (sources.cm):propset.sml]
[loading (sources.cm):lang.grm.sig]
[loading (sources.cm):prop.sml]
[loading (sources.cm):planner.sig]
[loading (sources.cm):planner.sml]
[loading (sources.cm):lang.lex.sml]
[loading (sources.cm):lang.grm.sml]
[loading (sources.cm):parser.sml]
[compiling (sources.cm):test.sml]
[code: 1618, data: 98, env: 665 bytes]
[library $smlnj/viscomp/parser.cm is stable]
[library $smlnj/MLRISC/MLRISC.cm is stable]
[library $SMLNJ-MLRISC/MLRISC.cm is stable]
[library $Graphs.cm(=$SMLNJ-MLRISC)/Graphs.cm is stable]
[library $smlnj/viscomp/execute.cm is stable]
[library $smlnj/internal/smlnj-version.cm is stable]
[library $smlnj/viscomp/x86.cm is stable]
[New bindings added.]
- Test.planX:
val it = SOME [(("movAD",[]),("movDE",[]))] : Planner.plan option
-

```

## 4 Implementation Details

In this section, we would be going over the implementation specific details of the solution that was described earlier in Section 2. In this implementation, states of the problem are of type facts, which are implemented as a set, and actions that allow transition from one state to another, are of type rules, which are a triple of (action \* facts \* facts), in essence, this represents (rule-name \*

pre-conditions \* post-conditions). We will start by explaining the helper functions that are used by the planner functions, and then explain the four main planner functions.

The signature of the file planner.sml is given below:

```

1 (* Domain independent planner *)
2
3 signature PLANNER =
4 sig
5
6   (* The knowledge base is anything ascribing to the PROPSET signature *)
7   structure KB : PROPSET
8
9   type facts = KB.t
10  type action = KB.P.t
11  type rule = action * facts * facts
12  type plan = action list
13
14  (* Used by the parser *)
15  val newRule: KB.P.t * KB.t * KB.t -> rule
16
17  (* Planner functions *)
18  val planDfs      : rule list * facts * facts -> plan option
19  val planAllDfs   : rule list * facts * facts -> plan list
20  val planBfs      : rule list * facts * facts -> plan option
21  val planAllBfs   : rule list * facts * facts -> plan list
22
23  val toString: plan -> string
24
25 end

```

## 4.1 Helper Functions

**member:** facts \* facts list  $\rightarrow$  bool

**Note:** States are represented as facts

**Requires:** true

**Ensures:** true if the input state is a member of the state list, false otherwise

This function takes a state and a list of states and returns true if the state is present in the states list, and false otherwise. This function is implemented using recursion. It has one base case and one Inductive Case.

**Base Case:** We check if the list is empty and return false, as the state cannot be present in an empty list.

**Inductive Case:** We have a list with at least one element. We use the PROPSET function equal to check if the state is same as the state that was passed in to the function, if true, then we return true, else, we call the member function recursively on the remaining list.

```
member(e, x::l) = if KB.equal(e)(x) then true else member(e, l)
```

**getfrontierhelper:** facts list \* facts list \* (facts \* action list)list \* action list \* facts \* facts list  $\rightarrow$  (facts \* action list)list

This function takes the following arguments in order: pre-conditions list , post-conditions list,

accumulator, rules list, current state, visited states list.

**Requires:**  $\text{Length}(\text{pre-conditions list}) = \text{Length}(\text{post-conditions list}) = \text{Length}(\text{rules list})$ , and that each element in the rules list corresponds to an element in the pre-conditions list and post-conditions list, in order in which they occur. For example, the preconditions and post-conditions are at index 0 of their respective lists, for a element in the rules list whose index is 0.

**Ensures:** Result is the frontier nodes obtained after applying the rules to the current state.

This function has one base case and one Inductive case.

**Base Case:** The base case is that all of the three lists, respectively the pre-conditions list, the post conditions list, and the rules list, are empty. In this case, we just return the accumulator.

**Inductive Case:** The Inductive Case is that all of the three lists, respectively the pre-conditions list, the post conditions list, and the rules list, at least have one element.

```
| getfrontierhelper (x::l,y::b,accum,act::act',init,visited) =
```

In this case, take the first element out of each list and do the following operations in order:

1. Using the diff function in the Propset, take the difference of init state and the element x from the preconditions list. And then apply the union function on the result and the element form the post conditions list(y).

```
| val newstate = KB.union (KB.diff(init)(x)) (y)
```

2. Using the member function, check if the resulting newstate from step 1 is a member of the visited states list
3. if the result of step 2 is true, then recursively call the getfrontierhelper on the remaining of the three lists: respectively the pre-conditions list, post-conditions list, and the rule list. All other arguments get passed in unchanged.

```
| if member(newstate,visited) then getfrontierhelper(l,b,accum,act',init,visited)
```

Else recursively call the getfrontierhelper on the remaining of the three lists: respectively the pre-conditions list, post-conditions list, and the rule list. All other arguments get passed in unchanged, except the accumulator, the accumulator is changed to Cons((state from step 1,[elem from rules list]), accumulator)

```
| else getfrontierhelper(l,b,(newstate,[act])::accum,act',init,visited)
```

What step 1 is effectively doing is that it is applying a rule to the current state, by removing the pre-conditions from the initial state and replacing them with the post conditions. Step 2 checks if we have already seen a similar state before, so we don't put a state in our frontier which we have already seen, as this could possibly cause an infinite loop in which our frontier is never empty and we keep cycling between the same states. Step 3 ensures that the function is called recursively on the rest of the input, while the tuple (newstate,[rulename]) is added to the accumulator/frontier if we did not visit the newstate before.

**getfrontier:** rule list \* facts \* (facts \* action list)list \* facts list  $\rightarrow$  (facts \* action list)list  
 This function takes the following arguments in order: rules list , current state, accumulator, visited states list.

**Requires:** true

**Ensures:** returns the corresponding frontier nodes for the given state, by applying the applicable rules in the rule list.

This function has one base case and one Inductive case.

**Base Case:** The base case is that the rules list is empty. In this case, we just return the accumulator.

**Inductive Case:** The Inductive Case is that the rules list at least has one element of the form (rule-name, preconditions, post-conditions).

```
| | getfrontier((a,pre,post)::l,init,accum,visited) =
```

We do the following operations on that element:

1. Using the PROPSET function subsetInstances, we get the list of substitutions which would be required to make the preconditions a subset of the current state. If subsetInstances throws an exception NoInstances, we handle it by saving the result of the function as NONE.

```
|         val subsetins = (SOME (KB.subsetInstances(pre)(init))
                           handle NoInstances => NONE)
```

2. After step one, the control flow of the function would depend on the value returned by subsetInstances which is saved in the variable subsetins. If subsetins = NONE, follow through step 3, if subsetins = SOME [ [ ] ] i.e an empty list of substitutions, follow through step 4, else, subsetInstances returned a non-empty list of substitutions, in this case follow through step 5.

3. if subsetins = NONE is true, it means that no such substitutions existed such that the preconditions could be made a subset of the current state. In this case we would just discard the rule we were working on and recursively call getfrontier on the rest of the rule list, with all other arguments unchanged.

```
| if (subsetins = NONE) then getfrontier(l, init, accum, visited)
```

4. if subsetins = SOME [ [ ] ], it means that the preconditions did not have any variables in them and they were a ground set which was already a subset of the current state. In this case we check if the post-conditions are already a member of the visited list.

**If member returns true,** we call getfrontier on the remaining rule list with all other inputs unchanged.

```
| if member(post,visited) then getfrontier(l,init,accum,visited)
```

**Else** we apply the rule on the current state, and get a newstate by taking the difference of the current state and the pre-conditions, and then taking a union of the result with the post-conditions. We then recursively call getfrontier on the remaining rule list with all other inputs unchanged except accumulator, the accumulator is changed to Cons((newstate,[rule-name]), accumulator )

```

1 | let
2 |     val newstate = KB.union (KB.diff(init)(pre)) (post)
3 | in
4 |     getfrontier(l,init,(newstate,[a])::accum,visited)
5 | end

```

5. In this case, subsetInstances returned a non empty list of substitutions that when applied to the preconditions, will make it a subset of the current state. We will now do the following operations on the list of substitutions returned by subsetInstances:

- (a) Use the PROPSET function apply, to apply the list of substitutions to the preconditions and post conditions. For this we will use the List.map function which will apply each substitution to the preconditions.

```

1 | val newinitialstates = List.map (fn x=> KB.apply(x)(pre))
2 |   (valOf(subsetins))
3 | val newfinalstates = List.map (fn x=> KB.apply(x)(post))
4 |   (valOf(subsetins))

```

- (b) Use the PROP function apply, to apply the list of substitutions to the rulename. For this we will use the List.map function which will apply each substitution to the rulename.

```

| val instantiatedrules = List.map (fn x=> KB.P.apply(x)(a)) (valOf
|   (subsetins))

```

- (c) Get the new frontier by calling the getfrontierhelper on the newinitialstates, newfinalstates, and instantiatedrules.

```

| val newfrontier = getfrontierhelper(newinitialstates,newfinalstates
|   ,[], instantiatedrules,init,visited)

```

- (d) Call the getfrontier function recursively on the remaining rule list, with all other inputs unchanged except the accumulator, the accumulator is changed by appending the result from step 3 at the beginning of the accumulator.

```

| getfrontier(l,init,newfrontier@accum,visited)

```

What step 1 is doing that it is doing that it is getting the list of substitutions which would be required to make the preconditions a subset of the current state. If the no such substitutions exist, subsetInstances would raise an exception NoInstances which is dealt with in step 3. If the preconditions and the current state both were ground, i.e they did not have any variables in them, and the preconditions were a subset of the current state, subsetInstances would return SOME empty substitutions list, which is taken care of in step 4. Otherwise, the substitutions list returned by subset instances is dealt with in step 5.

**planAllSolutions:** rules list \* facts \* facts \* (facts \* action list)list \* bool \* plan list \* facts list \* bool → plan list

This function takes the following arguments in order: (rules list , initial state, final state, frontier, flag,accumulator, visited states list,dfs flag)

**Requires:** true

**Ensures:** returns a list of all the plans which take us from the initial state to the goal state if any

exist, empty list otherwise.

This function has one base case and one Inductive case.

**Base Case:** The base case is that the rules list is empty. In this case, we just return the accumulator.

**Inductive Case:** The Inductive Case is that the rules list at least has one element . In this case, we do the following:

1. We check if the flag is set (true) and the frontier is empty then we return the accumulator.

```
| if flag andalso (frontier = []) then accum
```

If the flag is not set and the frontier is empty, follow through step 2, else the frontier is non empty, in this case follow through step 3.

2. In the case where the frontier is empty, first we check if the final state is already a subset of the initial state using the subset function from PROPSET. If true, we return an empty plan list.

```
| if (KB.subset(final)(init)) then []
```

**Else**, to get the frontier corresponding to the initial state, we call the getfrontier function on the following input and save the result.

```
| val front = getfrontier(rl,init,[],init::visited)
```

Then we recursively call planAllSolutions on all the same inputs except the frontier, flag, and the visited states list. The frontier is replaced with the result of getfrontier (front), the flag is set to true (as we have filled the frontier with the corresponding nodes of the initial state), and visited states list is replaced with Cons(initial state, visited states list), because we have seen the initial state now.

```
| planAllSolutions(rl,init,final,front,true,accum,init::visited,dfs)
```

3. In the case where the frontier is non-empty, it means that it has atleast one element of the form (current-state,path), where path is the respective path from the initial state that took us to the current-state.

```
| val ((current,path)::l)= frontier
```

In this case we check if the current-state is a subset of the final state, to determine if we have reached the goal state.

```
| if KB.subset(final)(current)
```

**If true** we recursively call planAllSolutions on all the same inputs except the frontier and accumulator. The frontier is changed by removing the current element, and the accumulator is changed by appending the path at the beginning of it.

```
| planAllSolutions(rl,init,final,l,flag,[path]@accum,visited,dfs)
```

**Else**

- (a) We get the corresponding frontier nodes of the current-state, by calling the `getfrontier` function, and use `List.map` to map the current-state's path to the path of the nodes that are returned as a result of `getfrontier`.

```
1 | val newfrontier = getfrontier(rl,current,[],current::visited)
2 | val newfrontier' = List.map (fn (x,y)=>(x,path@y)) newfrontier
```

- (b) If `dfs` was set (true) that means we have to do depth first search, so we recursively call `planAllSolutions` on all the same inputs except the frontier and the visited states list. The frontier is changed by appending the result of (a) at the beginning of the frontier, and the visited states list is replaced by `Cons(current-state,visited state list)`. By appending the new frontier at the beginning of the old frontier, we ensure that the order of search is depth-first, as the frontier list would behave as a LIFO stack.

```
| if dfs then planAllSolutions(rl,init,final,(newfrontier' @ l),flag
| ,accum,current::visited,dfs)
```

Else we are supposed to do breadth-first search, so we recursively call `planAllSolutions` on all the same inputs except the frontier and the visited states list. The frontier is changed by appending the result of (a) at the end of the frontier, and the visited states list is replaced by `Cons(current-state,visited state list)`. By appending the new frontier at the end of the old frontier, we ensure that the order of search is breadth-first, as the frontier list would behave as a FIFO queue.

```
| else planAllSolutions(rl,init,final,(l @ newfrontier'),flag,accum,
| current::visited,dfs)
```

What step 1 is doing is that it is checking if the flag is true and the frontier is empty then return accumulator. We are doing this because we explicitly set the flag to true when the frontier is empty in the first time we enter the function, if after that we see that the frontier is empty, that means that we have visited all nodes. Therefore we return the accumulator.

In step 2 we are adding the frontier nodes of the initial state to the frontier list. This happens only when we are entering the function for the first time with the frontier passed in as an empty list.

Step 3 deals with the case where the frontier is non empty, it checks if the state in the first element in the frontier is the goal state, if true, it adds it to the accumulator and calls `planAllSolutions` on the rest of the frontier recursively to find other possible solutions. Else it expands that node into its frontier nodes and adds those to the frontier. The nodes are added at the beginning or end of the frontier depending on if `dfs` was true or false.

## 4.2 Planner Functions

**planDfs:** `rule list * facts * facts → plan option`

**Requires:** true

**Ensures:** If a plan exists which takes us from the initial to the goal state, `planDfs` returns SOME plan, NONE otherwise.

In this function, we call the helper function `planAllSolutions`, which returns all solutions to the problem if any exist, or an empty list if there are no solutions. We pass in the `dfs` flag as `true`, so the the new frontier is always appended at the beginning of the existing frontier in the `planAllSolutions`, this way we will be able to achieve depth first search as the frontier would act as a LIFO stack. `flag` is passed as `false`, as it indicates that it is the frontier has not been filled with the frontier nodes of the initial state yet. After calling `planAllSolutions`, we case on the result, if the result is an empty list, we return `NONE` as no solutions were found, else we return `SOME` first element of the list.

**planAllDfs:** `rule list * facts * facts → plan option`

**Requires:** `true`

**Ensures:** If plans exists such that they take us from the initial to the goal state, `planAllDfs` returns the plan list containing all such plans, empty plan list otherwise.

In this function, we call the helper function `planAllSolutions`, which returns all solutions to the problem if any exist, or an empty list if there are no solutions. We pass in the `dfs` flag as `true`, so the the new frontier is always appended at the beginning of the existing frontier in the `planAllSolutions`, this way we will be able to achieve depth first search as the frontier would act as a LIFO stack. `flag` is passed as `false`, as it indicates that it is the frontier has not been filled with the frontier nodes of the initial state yet. We return the result of `planAllSolutions`.

**planBfs:** `rule list * facts * facts → plan option`

**Requires:** `true`

**Ensures:** If a plan exists which takes us from the initial to the goal state, `planBfs` returns `SOME` plan, `NONE` otherwise.

In this function, we call the helper function `planAllSolutions`, which returns all solutions to the problem if any exist, or an empty list if there are no solutions. We pass in the `dfs` flag as `false`, so the the new frontier is always appended at the end of the existing frontier in the `planAllSolutions`, this way we will be able to achieve breadth first search as the frontier would act as a FIFO queue. `flag` is passed as `false`, as it indicates that it is the frontier has not been filled with the frontier nodes of the initial state yet. After calling `planAllSolutions`, we case on the result, if the result is an empty list, we return `NONE` as no solutions were found, else we return `SOME` first element of the list.

**planAllBfs:** `rule list * facts * facts → plan option`

**Requires:** `true`

**Ensures:** If plans exists such that they take us from the initial to the goal state, `planAllBfs` returns the plan list containing all such plans, empty plan list otherwise.

In this function, we call the helper function `planAllSolutions`, which returns all solutions to the problem if any exist, or an empty list if there are no solutions. We pass in the `dfs` flag as `false`, so the the new frontier is always appended at the end of the existing frontier in the `planAllSolutions`, this way we will be able to achieve breadth first search as the frontier would act as a FIFO queue. `flag` is passed as `false`, as it indicates that it is the frontier has not been filled with the frontier nodes of the initial state yet. We return the result of `planAllSolutions`.

## 5 Discussion on decidability, soundness, and completeness

The planning problem is decidable. This means that, if a solution exists for any problem, it will be returned, and if no solution exists, the algorithm will identify this and return. This is possible because the search space in the planning problem is finite. The fact that the search space is finite



is important, as it will ensure that the program does not loop forever and reaches a point where all of the states have been seen and the program comes to a halt. To understand why the search space is finite, let's discuss the scenarios that could make the search space infinite and then deduce why our search space is finite.

The first scenario that could make the search space infinite is that we have an infinite number of rules, i.e the rule list provided to the program is infinite. In this scenario, when we call the `getfrontier` function for the first time to get the frontier nodes corresponding to the initial state, `getfrontier` wont return and go into an infinite loop, this would happen because the rule list would never become empty. However, this is not the case, because the rule list provided to the program is finite, therefore we can only have a finite number of frontier nodes corresponding to each node, thus the search space is finite.

The second scenario that could make the search space infinite is that if we don't have terminal states in our state space. A terminal state is a state where no more rules can be applied, i.e the state does not satisfy the preconditions for any rules. This would make the search space infinite as our frontier would never become empty and we would keep looping forever. However, this is not the case, because as our rule list is finite, and we are not adding visited nodes to the frontier, we would eventually reach a state where no rule in the rule list can be applied to a state, therefore that state would become a terminal state and it wont be added to the frontier. Thus our frontier would be finite, and we would either find a solution, or the frontier would become empty and we would identify that no solution was found.

Another scenario that could make the search space infinite would be if we had loops in our code. For example, if we had two rules in the rule list: rule1:  $a \rightarrow b$ , rule2:  $b \rightarrow a$ . If the initial state was either a or b, the program would keep looping between the two states forever. However, this is not the case, as when we visit a state, we add it to our visited states list and then the next time before adding a state to the frontier, we first check if that state is already present in the visited list. If it is was visited before, we don't add it. Therefore there would be no loops between states, therefore our search space would be finite, as the program will either find a solution or the frontier will become empty and we will identify that no solution was found.

Now since we have deduced that the search space is finite and the planning problem is decidable, this means that the planner functions are sound and complete. The functions are complete because as they are decidable, for any valid input if there exists a solution, the program will return the solution, else it will terminate and identify that no solution was found. The functions are sound because as we are avoiding loops, the search space is finite, and therefore the contrapositive of soundness would hold, i.e if there is no solution, then the program wont find a solution. Therefore the program would only find a solution if there exist a solution. Hence the program is both sound and complete

If we forgot to check for loops, it would affect decidability and completeness, but not soundness. The algorithm wont be complete as if it is given a input that has a solution but causes looping, our program would loop forever and wont terminate, therefore it wont return the solution and is incomplete. The algorithm wont be decidable as well, because the infinite looping would make the

search space act like an infinite search space when it is not, and the program wont be able to decide whether a solution exists or not. Therefore, it wont be decidable. However, the program would be sound, as for the inputs for which it does return an output, the output would be a valid solution.

The **completeness** theorems for the functions are:

**planDfs:**

For every initial state **init**: facts, goal state **final**: facts, and finite rule list **rl** : rule list, if there exists a sequence of rules **x**: plan, such that when applied to the initial state takes us to the goal state, then planDfs(rl, init, final) returns SOME x.

**planAllDfs:**

For every initial state **init**: facts, goal state **final**: facts, and finite rule list **rl** : rule list, if there exists a sequence of rules **x**: plan, such that when applied to the initial state takes us to the goal state, then planAllDfs(rl, init, final) returns a plan list containing all such plans.

**planBfs:**

For every initial state **init**: facts, goal state **final**: facts, and finite rule list **rl** : rule list, if there exists a sequence of rules **x**: plan, such that when applied to the initial state takes us to the goal state, then planBfs(rl, init, final) returns SOME x.

**planAllBfs:**

For every initial state **init**: facts, goal state **final**: facts, and finite rule list **rl** : rule list, if there exists a sequence of rules **x**: plan, such that when applied to the initial state takes us to the goal state, then planAllBfs(rl, init, final) returns a plan list containing all such plans.

The **soundness** theorems for the functions are:

**planDfs:**

For every initial state **init**: facts, goal state **final**: facts, and finite rule list **rl** : rule list, if planDfs(rl, init, final) returns SOME x, then x is a sequence of rules such that when applied to the initial state take us to the goal state.

**planAllDfs:**

For every initial state **init**: facts, goal state **final**: facts, and finite rule list **rl** : rule list, if planAllDfs(rl, init, final) returns a non-empty plan list, then the list has sequences of actions such that when applied to the initial state take us to the goal state.

**planBfs:**

For every initial state **init**: facts, goal state **final**: facts, and finite rule list **rl** : rule list, if planBfs(rl, init, final) returns SOME x, then x is a sequence of rules such that when applied to the initial state take us to the goal state.

**planAllBfs:**

For every initial state **init**: facts, goal state **final**: facts, and finite rule list **rl** : rule list, if planAllBfs(rl, init, final) returns a non-empty plan list, then the list has sequences of actions such that when applied to the initial state take us to the goal state.

## 6 Complexity analysis and heuristics

### 6.1 Complexity Analysis

#### 6.1.1 Complexities of Helper Functions

The complexities for the helper functions used in the main planner functions are given below:

**KB.subset:**  $O(n^2)$

The complexity of this function is  $n \times$  the complexity of the member function. The complexity of the member function is  $n \times$  the complexity of the P.eq function whose complexity is  $O(1)$  as it is performing a constant time equality check operation. Therefore the complexity of the member function is  $O(n)$ , and consequently, the complexity of KB.subset is  $n \times n$ , which is  $O(n^2)$ .

**KB.equal:**  $O(n^2)$

The complexity for this function is 2 times the complexity of the KB.subset function, which is  $2(n^2)$ . Therefore the complexity of KB.equal is  $n^2$ .

**KB.diff:**  $O(n^2)$

The complexity for this function is  $n \times$  the complexity of the KB.remove function. The complexity of KB.remove is  $O(n)$  as it goes through the entire list to find the element. Therefore the complexity of KB.diff is  $n \times n$ , which is  $O(n^2)$ .

**KB.union:**  $O(n^2)$

The complexity for this function is  $n \times$  the complexity of the KB.add function. The complexity of KB.add is  $O(n)$  as it uses the List.exists function, whose complexity is  $O(n) \times$  the complexity of the function provided to it, which in this case is P.eq, whose complexity is  $O(1)$ . Therefore KB.add is in  $O(n)$ , therefore KB.union is in  $n \times n$ , which is  $O(n^2)$ .

**P.apply:**  $O(n^2)$

Lets assume that the action supplied to P.apply has  $n$  variables, then the complexity becomes  $n \times$  complexity of P.apply. Complexity of P.apply depends on the size of the substitution passed in. Assuming that the substitutions would be the same as the number of variables or more, the complexity becomes  $n^2$ .

**KB.apply:**  $O(n^3)$

The complexity for this function is  $n \times$  the complexity of the P.apply function. Therefore the complexity of KB.apply is  $n \times n^2$ , which is  $O(n^3)$ .

**KB.subsetInstances:**  $O(n^4)$

The complexity for this function is  $\max(\text{complexity of KB.subset}, \text{complexity of ssInstances})$ . The complexity of ssInstances is : (Complexity of List.foldr  $\times$  complexity of KB.apply + complexity of remove + complexity of List.map  $\times$  complexity of P.compose) + Complexity of KB.instances.

Which is:  $(n \times (n^3 + n + n \times n^2)) + n^3 = n \times (n^3) + n^3 = n^4$

Therefore the complexity for subsetInstances is  $\max(n^2, n^4)$ , therefore it is in  $O(n^4)$ .

**member:**  $O(v)$

The complexity of the member function depends on the complexity of the visited state list, and the complexity of the KB.equal function. Lets suppose that the length of the visited state list is  $v$ , and the max number of elements in the pre or post conditions set is  $k$ , then the complexity of KB.equal would be  $k^2$ . Assuming that the size of the visited states list would be greater than  $k^2$ . The complexity becomes  $O(v)$ .

**getfrontierhelper:**

In this function,  $KB.diff$  works on an element from the instantiated preconditions list. We assume that the length of this element is  $k$ .  $KB.union$  works on the result of the diff function called on the precondition and the initial state, let the length of the resulting element be  $j$ . Then we have the following recurrence:

$$Wgetfrontierhelper(n) = \begin{cases} a0, & \text{if } n = 0 \\ a1 + WKB.diff(k) + WKB.union(j) + getfrontierhelper(n-1) & \text{otherwise} \end{cases}$$

#### **getfrontier:**

Let the size of the initial state be  $b$ , visited list be  $c$ , rule list be  $n$ .

To simplify the calculations we assume that on average the preconditions and postconditions are of the same size, let this size be  $d$ .

Lets also for the sake of simplicity, assume that each rule can have a maximum of  $q$  variables and  $k$  constants. Then the size of the substitutions returned by subset instances would be:  $nk^q$

$$Wgetfrontier(n) = \begin{cases} a0, & \text{if } n = 0 \\ a1 + WKB.subsetInstances(b) + \max(Wgetfrontier(n-1), (Wmember(c) + Wgetfrontier(n-1)), (nk^q) \times 2 * WKB.Apply(d) + (nk^q) \times WKB.P.apply(q) + Wgetfrontierhelper(nk^q) + Wgetfrontier(n-1))) & \text{otherwise} \end{cases}$$

#### **planAllSolutions:**

In this function, let the size of, frontier be  $n$ , final state be  $j$ , rule list be  $r$ , result of getfrontier at any given call be  $f$ , old frontier before adding the result of getfrontier be  $m$ . Let the symbol denoting the flag be  $f$ .

$$planAllSolutions(n, f) = \begin{cases} a0, & \text{if } n = 0 \text{ and } f = true \\ a1 + \max((WKB.subset(j) + Wgetfrontier(r) + WplanAllSolutions(f)), (WKB.subset(j) + WplanAllSolutions(m-1)), (getfrontier(r) + planAllSolutions(m-1 + f))) & \text{otherwise} \end{cases}$$

### **6.1.2 Complexity for the Planner Function:**

#### **planDfs:**

Let the empty frontier be  $f$ , and the rule list be  $r$ .

$$WplanDfs(r) = \begin{cases} a0 + WplanAllSolutions(f, false) \end{cases}$$

## **6.2 Heuristics**

A simple heuristic that does not prune the search space is **Depth-limited search**. In Depth-Limited Search, we explore the graph to a given depth, if a solution is found within that depth

limit, we return that solution, else we identify that no solution could be found. This heuristic can be tricky in implementing as it would be hard to decide a depth limit for a given problem, what if we decide a limit that is too low? or too high? If its too low we could miss the chance of finding a solution if it existed at a deeper depth, if it is too high then the program might take too long to give an output. Depth-limited search affects the soundness, as the program is no longer sound because if the program gives No solution on an input, it does not necessarily mean that there was no solution for the valid input, it could also mean that a solution could not be found within the depth limit. Therefore, soundness is affected. However, Completeness is not affected as the search space is still finite and the program would terminate if no solution's is found within the depth limit, or return a valid solution.

A heuristic that prunes the search space would be to use a simple heuristic function that assigns a non-negative integer to every state. This non negative integer would represent the distance to the goal from the current state. This is called **Greedy Search**. This wont affect completeness as the search space is still finite and the program would terminate if it has visited all states and no solution's is found, or return a valid solution. This also wont affect soundness, as if the program returns a valid solution that means it found it while traversing the graph, and if it returns No solution, it means that all nodes were explored and no solution was found. The function that would assign costs to nodes would be left for the user to supply.

## 7 References

1. The Planning Process [Digital Image](n.d.). Retrieved December 9, 2018 from <https://web.stanford.edu/class/cs227/Lectures/lec16.pdf>
2. Di Caro, Gianni A. (n.d.). General Graph Search. Retrieved December 9, 2018 from <https://web2.qatar.cmu.edu/gdicaro/15381/slides/381-F18-2-SearchProblems-and-UninformedSearch.pdf>