Planning

What we did?

We did stick to the provided template and followed the guidelines provided in our implementation. Basically, we follow the recursion as such and we check for maximum depth to account for iterative deepening only in the select-subgoal function. If we reach the solution before we hit maximum depth we return the solution if not we return nil.

Choosing the sub goal:

We observed that (as stated in the hint), while you can pick any possible sub goal it would be easier to arrive in a solution (or fail soon) if you pick a sub goal from the operator with the least open preconditions. So, we introduced a new function to return the open precondition count for every operator in the plan and we picked the one with the least count. This way we meet all the preconditions for goal first and move our way back.

Choosing the operator:

Our initial approach was to try each operator in the plan which meets a specific precondition, then try the operators from the template. When we went for this approach, we noticed that the search space increased because, we ended up adding existing operators in the plan again (when we chose from the template) with a different unique symbol slowing down the search. To handle this situation we introduced a function to check if an operator already exists in the plan before we added the operator to the plan (This should have been obvious, but was not when we started).

Speed:

For reason I couldn't explain, we got better performance by staying away from the built hash table (for all-operators) and implementing it going over all possible operators (which sounds stupid). Our implementation did return in a search depth of one less than what is given in the example (specification).

Three block world Sussman's anomaly runtime – 17.5 seconds

Two block world runtime – .015 seconds

What we tried:

We tried to study the scalability of the implementation, for this we took the three-block implementation and gave it a difficult start condition (b on a, c on b and b on table) which was just the reverse of the goal condition. Our initial attempt was to just run the implementation and see how long it takes for it to come up with a plan. After couple of hours of run we went to a search depth of 16 without any luck. We lost patience at this point and thought of improving the performance of our implementation. Our first change the all operators function to pick operators from the hash table, we were not successful in improving the performance with this change. So, we started considering the reachable function.

What we changed with the reachable:

Our initial approach was based on noticing the association list is by itself a list of edges, and if we can build a graph with these edges we can use DFS to see if “to” is reachable from “from”. While the approach would yield asymptotically optimal runtime, we soon realized that most of the association lists generated for the problems we were trying to solve had a maximum length within 100, so the graph build time and search though it took most of the time offering no improvement in performance.

We next tried to use some hash tables and keep track of all the nodes we find from “from” and nodes we find from “to”. Again, all the hash tables took most of the overhead and we only observed drop in performance.

What worked:

So, we finally settled a simple version of reachable which keeps track of all the nodes reachable from “from” and chooses the next “from” from this list. This seems to offer better performance while the problem scales.

For the problem we wanted to test on scalability, we found that with an iterative deepening of 1, it was hard for us to get to the solution in short time. So we changed the search depth to 10, and we were able to get the solution, we are including just the ordering of the solution.

```
orderings:
#[ (G4837721)C-A-TO-TABLE -> (G4837719)A-TABLE-TO-B ]
#[ (G4837723)C-B-TO-TABLE -> (G4837719)A-TABLE-TO-B ]
#[ (G4837720)B-TABLE-TO-C -> (G4837719)A-TABLE-TO-B ]
#[ (G4837723)C-B-TO-TABLE -> (G4837720)B-TABLE-TO-C ]
#[ (G4837724)B-A-TO-TABLE -> (G4837722)C-TABLE-TO-A ]
#[ (G4837721)C-A-TO-TABLE -> (G4837720)B-TABLE-TO-C ]
#[ (G4837724)B-A-TO-TABLE -> (G4837719)A-TABLE-TO-B ]
#[ (G4837723)C-B-TO-TABLE -> (G4837724)B-A-TO-TABLE ]
#[ (G4837724)B-A-TO-TABLE -> (G4837720)B-TABLE-TO-C ]
#[ (G4837724)B-A-TO-TABLE -> (G4817572)GOAL ]
#[ (G4817571)START -> (G4837724)B-A-TO-TABLE ]
#[ (G4837723)C-B-TO-TABLE -> (G4837722)C-TABLE-TO-A ]
#[ (G4837723)C-B-TO-TABLE -> (G4817572)GOAL ]
#[ (G4817571)START -> (G4837723)C-B-TO-TABLE ]
#[ (G4837722)C-TABLE-TO-A -> (G4837721)C-A-TO-TABLE ]
#[ (G4837722)C-TABLE-TO-A -> (G4817572)GOAL ]
#[ (G4817571)START -> (G4837722)C-TABLE-TO-A ]
#[ (G4837721)C-A-TO-TABLE -> (G4817572)GOAL ]
#[ (G4817571)START -> (G4837721)C-A-TO-TABLE ]
#[ (G4837720)B-TABLE-TO-C -> (G4817572)GOAL ]
#[ (G4817571)START -> (G4837720)B-TABLE-TO-C ]
#[ (G4837719)A-TABLE-TO-B -> (G4817572)GOAL ]
#[ (G4817571)START -> (G4837719)A-TABLE-TO-B ]
#[ (G4817571)START -> (G4817572)GOAL ]
```

Time taken:

Our implementation of reachable for this problem	- 36 seconds
Provided implementation of reachable	- 43 seconds

Our implementation of reachable should be able to offer better performance for bigger problems which require large search space.