Automated Cooking With Planning

CSC 520 Final Project Report
Abir Majumder
Alex Nahapetyan

1 Project Description

1.1 The Problem and Why It Matters

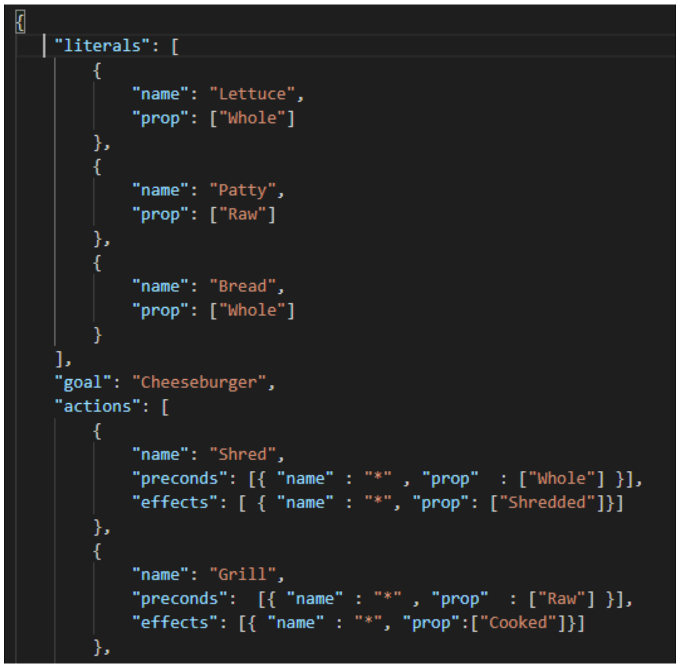
Cooking or preparing dishes is one of the most essential skills for self-nourishment. The process of cooking, however, is not a skill that everybody is born with. Cooking requires proper knowledge of time management, supply management, and, of course, knowing the right recipe structure. Some people don't want to be bothered with crafting a dish that may or may not taste good to them. They may not have the time nor the skills to cook. Thus, this project is designed to address these issues by creating a planning bot that automates the cooking process. This saves the time and hassle for someone to create their own dishes. A bot can make the dish for them.

1.2 The Plan and Justification

We will be considering an agent that is modeled as a Planning and Scheduling problem. More specifically, we will be taking the Planning Graph approach where an agent will have to evaluate states, possible actions, and effects of actions iteratively until a goal is reached. Recipes are a great complement towards this type of problem because it is characterized by a set of ingredients that may change state through the effects of actions such as frying, shredding, or mixing those ingredients into new ingredients which can change to other ingredients needed for the final product.

2 Project Details

This project uses a Planning agent utilizing an abstraction of the planning graph data structure and the GraphPlan algorithm in Python to construct recipes based on available ingredients, available actions, and the end goal. The ingredients, actions, and goal are to be specified in a JSON object that is similar to a PDDL style format. This is seen in the following image.



```
{
    "literals": [
        {
            "name": "Lettuce",
            "prop": ["Whole"]
        },
        {
            "name": "Patty",
            "prop": ["Raw"]
        },
        {
            "name": "Bread",
            "prop": ["Whole"]
        }
    ],
    "goal": "Cheeseburger",
    "actions": [
        {
            "name": "Shred",
            "preconds": [{ "name" : "*" , "prop"  : ["Whole"] }],
            "effects": [ { "name" : "*", "prop": ["Shredded"]}]
        },
        {
            "name": "Grill",
            "preconds":  [{ "name" : "*" , "prop"  : ["Raw"] }],
            "effects": [{ "name" : "*", "prop":["Cooked"]}]
        },
```

The “*” in preconditions and effects of actions are meant to indicate that it can accept any name as long as it has the matching properties. The Python program includes a JSON parser that translates this information upon startup. You can specify your own custom input just as long it is formatted correctly. The codebase is split into multiple classes: Graph, World, Action, State.

The *Graph* class is the overarching class that contains everything. It has an array of *Worlds*, an array containing all the actions the system can take, and the end goal.

The *World* class represents the layers of our planning graph implementation. Each *World* contains a set of literals (or *States*), an array of possible *Actions*, an array of action mutexes, and an array of literal mutexes. The index of a *World* class in the *Graph* specifies essentially two layers of the planning graph.

The program essentially does a GraphPlan search after it reads and translates the JSON parser. This includes calculating mutexes and recursively specifying a solution. After the algorithm finishes, assuming that a plan can be extracted in the first place, Graph Plan will return and print the sequence of steps to make the final food product.

3 Project Results

3.1 Experiment Setup

All Experiments were run using the Google Colab online notebook with 25GB of RAM. This was done to remove computational power as a bottleneck as much as possible while still altering us if something in the algorithm needs to be optimized.

3.2 First Run

The first batch of tests were run against ***naive_planer.py*** against knowledge base files of size 1KB, 2KB, and 4KB (input1.json, input2.json, input3.json).

3.2.1 Prediction

First run was conducted on a generic GraphPlan Algorithm that keeps track of all of the mutexes. In theory this would cause serious problems due to the program having to go through all the permutations of different states. In the worst case scenario, we will have a C($\boldsymbol{n}$,2) complexity, where $\boldsymbol{n}$ is the number of maximum states present in the program. Hypothetical, given small enough input, this wouldn't matter, however, if too many states are present in the knowledge base and they can be achieved given the initial conditions, the program would run out of memory.

3.2.2 Result

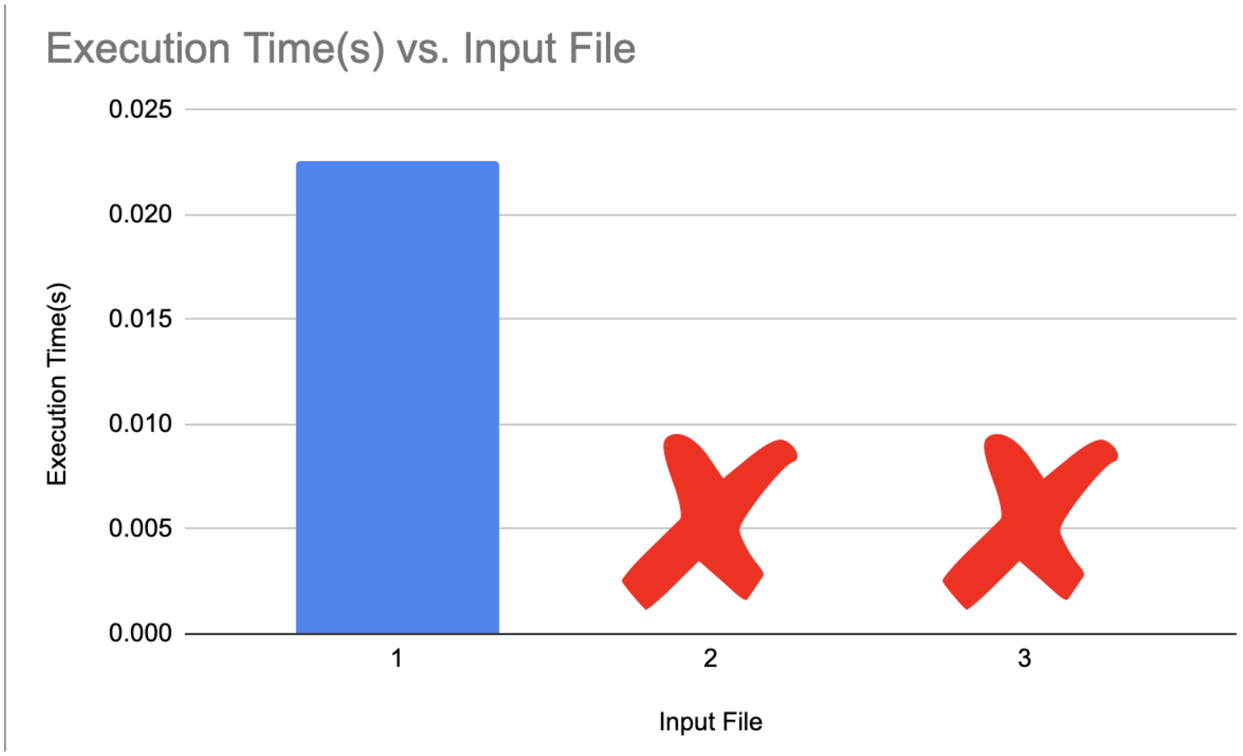


Figure 1

As you can see from Figure 1, while the first input file (***input1.json***), which has the bare minimum requirements for the goal to be achieved, the other 2 files crash due out-of-memory error from Google Colab.

3.3 Optimization Plan

We can make a few key observations about our specific problem. The most important mutexes to keep track of are action mutexes. And as long as ingredients are already carried over to the next layer via No-Op, we never have to care about precondition layers, since ingredients are replaced with new ones once an action is applied to them.

With this in mind, what we can do is remove the portion of the code that handles mutex calculation for precondition mutexes. And here are our new results.

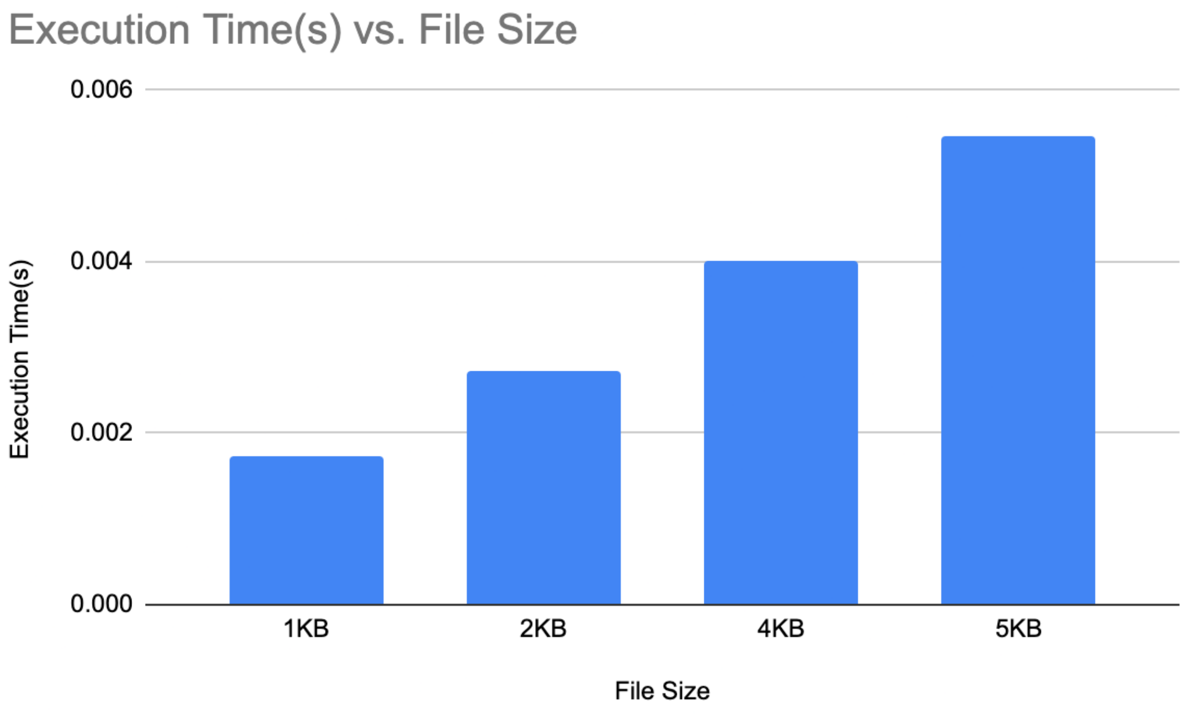


Figure 2

From Figure 2 we can see that this implementation still manages to run, and even reduces the time executed for the only test case (input1.json) that ran before. To better illustrate what is causing the increase in runtime, Figure 2 is plotted against knowledge base size (inputX.json file size).

3.4 Further consideration

Further performance tweaks for agents that were not considered in this project due to time constraints include.

3.4.1 Backend/Frontend architecture

Instead of dynamically trying to calculate all of the possible actions to take, and all of the mutexes, the program could leverage a prolog backend to resolve all of those queries. Implementing the knowledge base in prolog would have been simpler and the backend would run much faster then the JSON parser + Python code used in our setup.

3.4.2 Pre-processing

Before starting a graph plan, the knowledge-base could be per-processed by going through all “end products” (things that do not have actions that can act on them) and removing actions tied to them, as well as the items themselves out of the knowledge base. In theory this should reduce you down to a knowledge base on the side of inputX.json, however the per-processing itself might come at a hefty computational cost.

4 Demo

A demo of our agent can be found in the code as ***demo.py***; we also posted a video on youtube.