States and Planning Project Report

Alex Reigle

CEG 6850 Foundations in A. I.

Wright State University

Instructor: Dr. Nikolaos Bourbakis

December 10, 2021

Abstract

A windows command terminal application has been developed using an application of the topics taught in the Foundations in Artificial Intelligence course and textbook. This report provides an overview of the methods used to develop the constituent programs as well as the theoretical derivations that lead to their implementation. Flow charts have been created to provide a visual description of the algorithmic decisions being made. Selected results have been provided to demonstrate the capability and computational efficiency of the application.

Table of Context

Introduction

Sorting is a topic of much interest to the scientific community. A minute edge in sorting algorithm efficiency may be save companies millions of dollars in wasted processing time and other delays. Likewise, any logical flaws could result in catastrophe. For this reason, the improvement of sorting algorithms are always sought after by industry and investigated by engineers.

The objective of this project is to plan an optimal block sorting algorithm given a set of two generic input states and two generic output states. The least number of state changes possible is desired. In the following sections, I lay out the derivation of the plan and algorithm for the program provided as a solution to this project. I will the discuss the methodology of implementation as well as discuss how the algorithm achieves the end results before illustrating the progression through selected examples. Finally, I will discuss project outcomes and discuss the observed performance of the project.

For the algorithm developed, I have restricted the actions of the program to those defined within the universe of discourse (UoD). Those actions being as follows: Pick-up, Put-down, Stack, Unstack, Move, and No-op (No operation). I also restrict the UoD to the following pre-defined state descriptors: Above, On, Clear, Table. Using the defined UoD, the given definitions of the action designators and the allowed state descriptors, I show that I am able to derive the indexed position of any desired block. By defining these actions designators and state descriptors, I am able to define a plan to achieve the next intermediate state. The next intermediate state, in this program, is defined as one of a subset of the goal state the changes throughout the execution of the programs to iterate the block stacks closer to the final goal state – in terms of number of blocks in the location of their goal state, with a priority weighting towards the blocks closest to the table. The results of my algorithm are discussed

below. Additionally, the details of the methodology and logical structure are described below along with a demonstration of the progression through the algorithm via illustration.

Methodology

In the development of this program I considered the given dataset. I was given a UoD which contains only blocks {a, b, c, d, e, f, g, h, i, j, k, l, m, n}. The state descriptors given are Above, On, Clear, and Table. The actions (or functions) given are Pick-up, Put-down, Stack, Move, Unstack, and finally No-Op. The UoD also consists of two generic locations, L1 and L2, as well as two arms, ARM1 and ARM2, which can operate on the stacks of blocks to move or reconfigure them. I assume that the usage of the Clear state descriptor on an arm that is not holding a block will return true (for example, Clear(ARM1[0])==TRUE). I derive two Markov complete programs. The first sorts the two block stacks into a single block stack which can be operated on by the second program. The second program removes the blocks on the top on the stack at one location and places them in the other, until both stacks resemble the goal states.

I begin with the initial database to resolve an additional state designator that will be able to identify if a block, x, is located at a location, Li. This is function will be used in the subsequent derivation as follows: *bool = find(x, Li).*

$\forall x \text{Block}(x) \rightarrow a(x) \vee b(x) \vee c(x) \vee d(x) \vee e(x) \vee f(x) \vee g(x) \vee h(x) \vee i(x) \vee j(x) \vee k(x) \vee l(x) \vee m(x) \vee n(x)$	Δ1
$\forall x \text{Block}(x) \rightarrow L1(x) \vee L2(x)$	Δ2
$\forall x \forall z \forall i \text{ Block}(x) \wedge \text{Arm}(i) \wedge \text{Action}(z) \rightarrow \exists y \text{ Block}(y) \wedge \text{Arm}(i) \wedge \text{Action}(z)$	Δ3
$\forall x \forall z \forall i \text{ Block}(x) \wedge \text{Arm}(i) \wedge \text{Action}(z) \rightarrow \exists y \text{ Block}(x) \wedge \text{Arm}(i) \wedge \text{Action}(y)$	Δ4
$\forall x \text{Block}(x) \wedge (L1(x) \vee L2(x) \rightarrow a(x) \wedge (L1(x) \vee L2(x)) \vee b(x) \wedge (L1(x) \vee L2(x)) \vee c(x) \wedge (L1(x) \vee L2(x)) \vee \ldots$ (etc.)	1, 2

where, line 5 allows me to derive the following production system:

a(x) ∧(L1(x) → L(a(x))

b(x) ∧(L1(x) → L(b(x))

c(x) ∧(L1(x) → L(c(x))

d(x) ∧(L1(x) → L(d(x))

…

This sequence continues and holds true for each block in the UoD and will be replaced from hence forward with *find(x, Li)*, to be used as a state designator to identify a blocks' existence in a given location.

With this state designator abbreviation in place, I am now prepared to derive the first program developed in this project. The program is initialized using integers to indicate index (i), state (s), and a place holder to indicate the intended location of the final state (GL). The input of goal states are combined into a single stack prior to the initialization of program 1, this will be represented below as 'goal'. The first production system is defined as follows:

[(s=0) ∧ (i=0) ∧ find(goal[i], L1) → GL = L1,

(s=0) ∧ (i=0) ∧ find(goal[i], L2) → GL = L2,

Clear(goal[i]) ∧ Clear(ARM1[0]) → Pick-up(Clear(goal[i]), ARM1[0]),

ℸClear(goal[i]) ∧ ℸClear(ARM1[0]) → Pick-up(Clear(GL), ARM1[0]),

ℸClear(ARM1[0]) ∧ ℸ(ARM![0] = goal[i]) →Pick-up(Clear(GL), ARM2[0]),

ℸClear(ARM1[0]) ∧ ℸ(ARM![0] = goal[i]) ∧ find(ARM1[0], GL) → Unstack(Clear(GL)),

ℸClear(ARM1[0]) ∧ ℸ(ARM1[0] = goal[i]) ∧ find(ARM1[0], GL) ∧ ℸOn(Clear(GL)) → Move(ARM1[0], ℸGL),

ℸClear(ARM1[0]) ∧ ℸ(ARM1[0] = goal[i]) ∧ ℸfind(ARM1[0], GL) → Stack(ARM1[0], Clear(ℸGL)),

ℸClear(ARM1[0]) ∧ ℸ(ARM1[0] = goal[i]) ∧ find(ARM1[0], ℸGL) → Put-down(ARM1[0]),

(ARM1[0] = goal[i]) ∧ ℸ(GL = goal[0:i]) ∧ Clear(ARM2[0]) → Pick-up(Clear(GL), ARM2[0]),

(ARM1[0] = goal[i]) ∧ (ARM2[0] = Clear(GL)) → Pick-up(Clear(GL), ARM2[0]),

(ARM1[0] = goal[i]) ∧ (ARM2[0] = Clear(GL)) → Unstack(Clear(GL)),

(ARM1[0] = goal[i]) ∧ ℸClear(ARM2[0]) ∧ find(ARM2[0], GL) → Move(ARM2[0], ℸGL),

(ARM1[0] = goal[i]) ∧ ℸClear(ARM2[0]) ∧ find(ARM2[0], ℸGL) → Stack(ARM2[0], ℸGL),

(ARM1[0] = goal[i]) ∧ ℸClear(ARM2[0]) ∧ (Clear(ℸGL) = ARM2[0]) → Putdown(ARM2[0]),

Clear(ARM1[0]) ∧ find(ARM1[0], ℸGL) → Move(ARM1[0], GL),

Clear(ARM2[0]) ∧ find(ARM2[0], ℸGL) → Move(ARM2[0], GL),

(GL = goal) ∧ Clear(ARM1[0]) ∧ Clear(ARM2[0]) → No-Op]

This program sorts the blocks into a single stack the matches the goal stack. The ‘goal’ stack is a combination of the two final states input to the application.

The second program I have developed is also Markov complete. It takes the output of the first program, above, and sorts the single stack of blocks into the two block stacks that were input as the final state for the Uod. Here, I introduce a variable, sG, which is the stack definition for the final state of the stack that is not at the goal location, GL, of the previous program. This program is defined as follows.

¬(¬GL = sG) ∧ Clear(ARM1[0]) ∧ find(ARM1[0], GL) → Pick-up(Clear(GL), ARM1[0]),

¬(¬GL = sG) ∧ ¬Clear(ARM1[0]) ∧ find(ARM1[0], GL) ∧ (ARM1[0] = Clear(GL)) → Unstack(Clear(GL)),

¬(¬GL = sG) ∧ ¬Clear(ARM1[0]) ∧ ¬(ARM[0] = Clear(GL)) ∧ find(ARM1[0], GL) → Move(ARM1[0], ¬GL),

¬(¬GL = sG) ∧ ¬Clear(ARM1[0]) ∧ find(ARM1[0], ¬GL) → Stack(ARM1[0], Clear(¬GL)),

¬(¬GL = sG) ∧ ¬Clear(ARM1[0]) ∧ find(ARM1[0], ¬GL) ∧ (Clear(¬GL) = ARM1[0]) → Put-down(ARM1[0]),

¬(¬GL = sG) ∧ Clear(ARM1[0]) ∧ find(ARM1[0], ¬GL) → Move(ARM1[0], GL),

(¬GL = sG) →No-Op]

This program concludes with the two states, L1 and L2, reflecting the desired output states as defined by user input. The behavior of the programs developed are shown in the flowchart in Fig. 1 and Fig. 2, below.

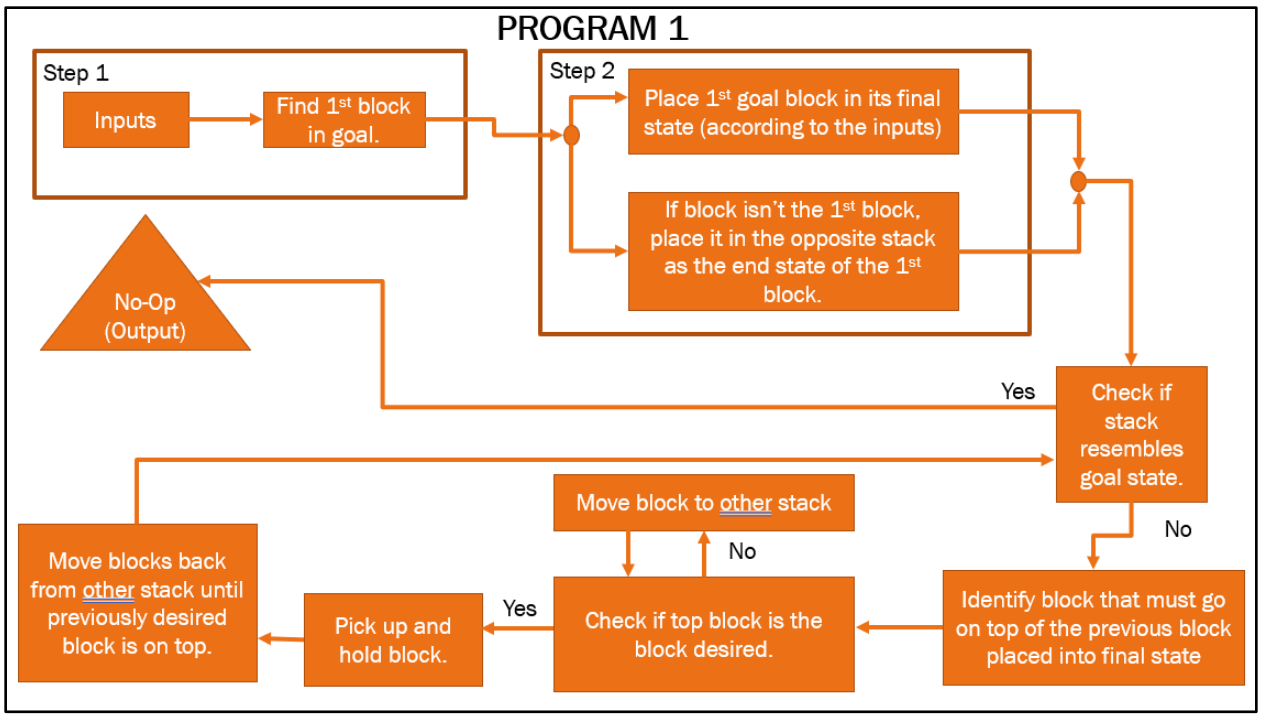

Figure 1: A flow chart showing the progression of my developed program 1. This is the program that makes the decision of which location would be the quickest to reconstruct the goal stack on.

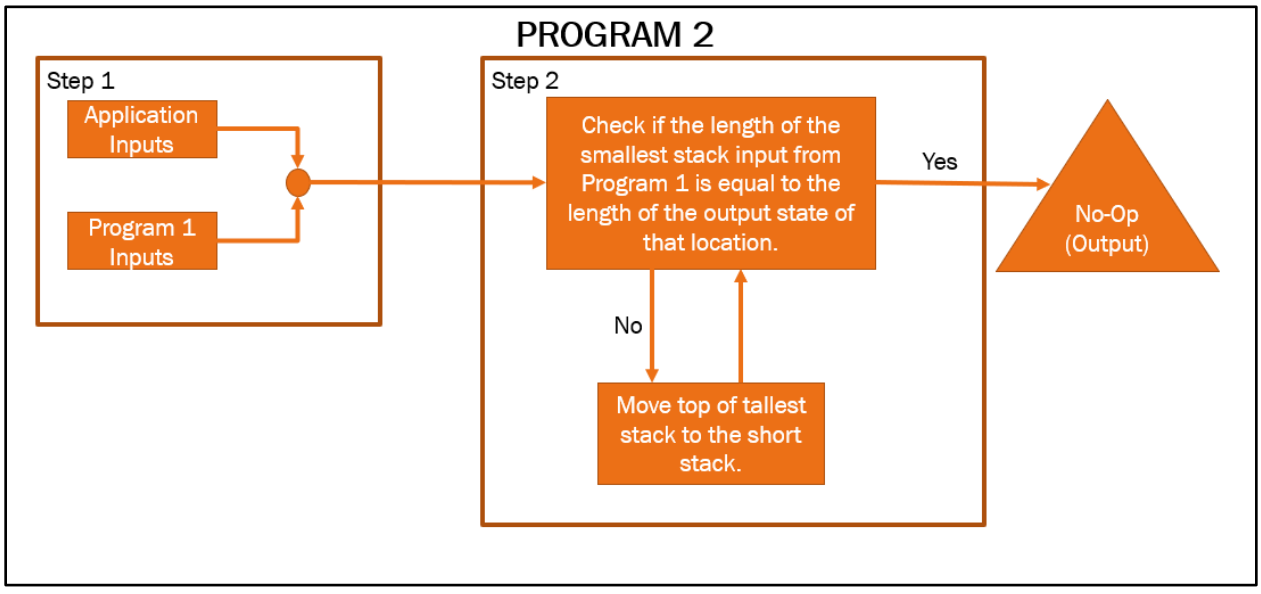

Figure 2: A flow chart showing the progression of my developed program 2. This program turns the one stack output from Program 1 into the two stacks defined as the final state.

The results of my algorithm are promising. I show, below, that the sorting method achieves the desired outcome and does so sufficiently quickly. While the step count reflects a large number, greater than the $O(n^2)$, the iterative process operates closer to O(n log(n)) time complexity. For example, given the inputs shown in Table 1, below, I am able to reach the final goal states in 470 steps.

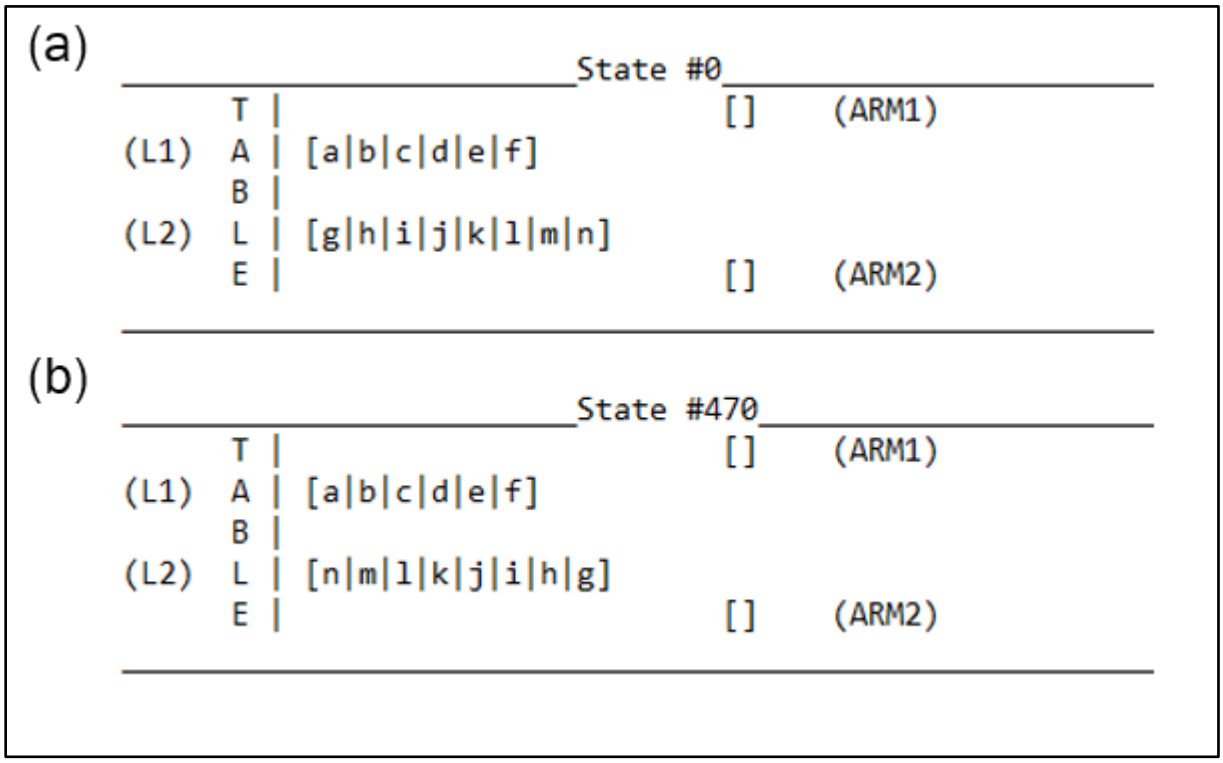

Table 1: Example 1 of successful results. (a) Input states L1='abcdef', L2='ghijklm'. (b) Output states L1='abcde', L2='nmlkjih'.

I show another example in Table 2, below. This shows that that algorithm I have developed sorts the near-worst-case input scenario in 596 steps. The example shown in Table 1 shows the developed algorithm accepts outlier cases that do not require operation to rearrange an input state.

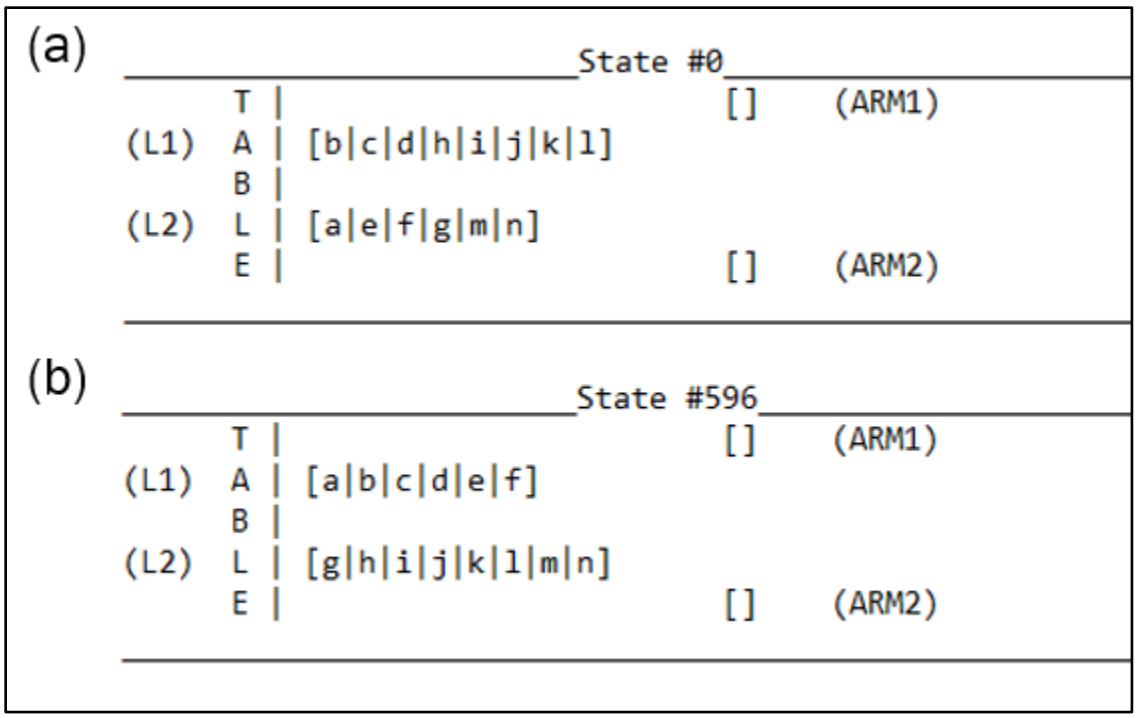

Table 2: Example 2 of successful results. (a) Input states L1='bcdhijkl', L2='aefgmn'. (b) Output states L1='abcdef', L2='ghijklmn'.

I believe, based on experimental results, that the programs I have developed work well and achieve a near-optimal sorting speed, and relies on logical structure that ensures good performance.

Conclusion

In this project, I resolved natural language into Predicate Calculus and used the axioms taught in this course generate a plan that I was able to implement as a C++ program. Using induction, I decide on the best location to initialize my sorting program with and the most efficient next state with which to progress the algorithm. The produced program implements the concepts discussed in the course and does so in a manner that results in a quality sorting algorithm.

References

1) Genesereth, Michael R., and Nils J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann, 2012.