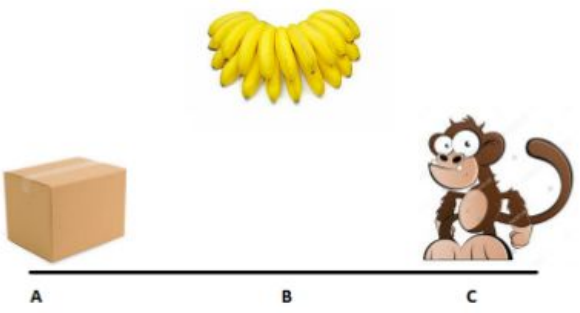
Planning

Hamit Kavas | Elif Hangül | Jonatan Koren



B.1. Monkey and bananas Problem

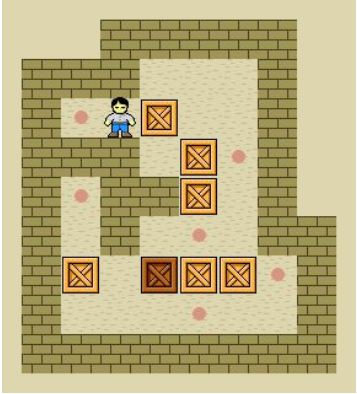
In this part of the homework, we wanted to keep our solution as simple as we can. We basically want to move the box under the banana, which will be followed by monkey's climbing the box and grab the banana. We check the monkey's vertical location, which shows if the monkey is on the floor or not with "on-floor" variable. Similarly, the monkey can only grab banana while "onbox" variable is True.
We only used objects below:

```
(location ?x)
(at ?x ?x)
(on-floor)
(hasbananas)
(onbox ?x)
)
```

To satisfy these constraints, we have used four actions;

- "goto" to move monkey.
- "climb" to put monkey onto a box with "onbox".
- "push-box" to move a box.
- "grab-bananas" to change state "hasbanas" to True. This is variable that shows we reached to the goal.

B.2. Sokoban



a) Player is limited to game board size - we have set the game board as follows:

```
;====================
;The initial puzzle ---> 5X4:
;====================
;x    1 2 3 4
;y
;1    . . # .
;2    . . . $
;3    $ . . .
;4    X @ $ X
;5    . . X .

;==========
; LEGEND:
;==========
;     # = Wall
;     . = Empty
;     @ = Player
;     $ = Box
;     x = Goal
```

Our domain file does not depend on the problem file and can generate solution for different given problems:
The predicates are has_player that indicates if there is a player, has_box that indicates if there's a box, neighbor that indicates position and neighbor position, neighbor_space that indicates about position and two spaces away from position, has_wall that tells if there is a wall, and the teleport.

There are Three actions that solves the problems:

- move-player: moves player from position to a new position if there is an empty neighbor without a box
- push-box: pushing box from one position to another position: check if there is an empty space 2 spaces away for each move in order to move the box
- teleport: checks if there's a player without a box and wall in the new position he wants to teleport himself and if the action is legal (only 2 times teleportation)

For the problem, our objects are defining the available positions on the board as well as the number of teleportation. we defined 2 possible teleportation for each given problem.
In the initial state we define all the possible neighbors of the player in each possible cell he can be at a time. Also we define all the possible 2 spaces from the player in each position he can be in a time. The neighbor is calculated for the move action of the player without a box, and the 2 spaces calculation is done for move action of a player with the box in order to understand the possibility of 2 moves away.
Also we define the position of the player, 3 boxes, wall and teleportation possibilities.

b) A general Domain.pdll file is defined. Input puzzle is given as:

"python sokoban.py -i benchmarks/sasquatch/level1.sok"

Based on the puzzle given as an argument, sokoban.py parses the puzzle and produces a Problem.pddl file which includes the states of the board for solving the puzzle. This file dynamically changes based on the input board. After creating the Problem.pdll, the sokoban.py calls fast downward solver with the command line:

"python fast-downward.py --alias lama-first --plan-file myplan.txt Domain Problem"

automatically to produce the output file myplan which includes the planned path and prints out this path on the console.

c) To give every puzzle inside of the benchmarks/sasquatch as an input for the pdll solver automatically, sokoban.py is changed to accepts two kinds of argument. If the file is called as:

"python sokoban.py -i benchmarks/sasquatch/level1.sok"

with just one puzzle path, then the solver solves just that puzzle. If the file is called as:

"python sokoban.py -i benchmarks"

the code will iterate over all the puzzles in benchmarks/sasquatch folder, create a Problem.pdll file and produce a solution for them in ./Greedy_BFS_Plans/ directory. For every puzzle below code will automatically called:

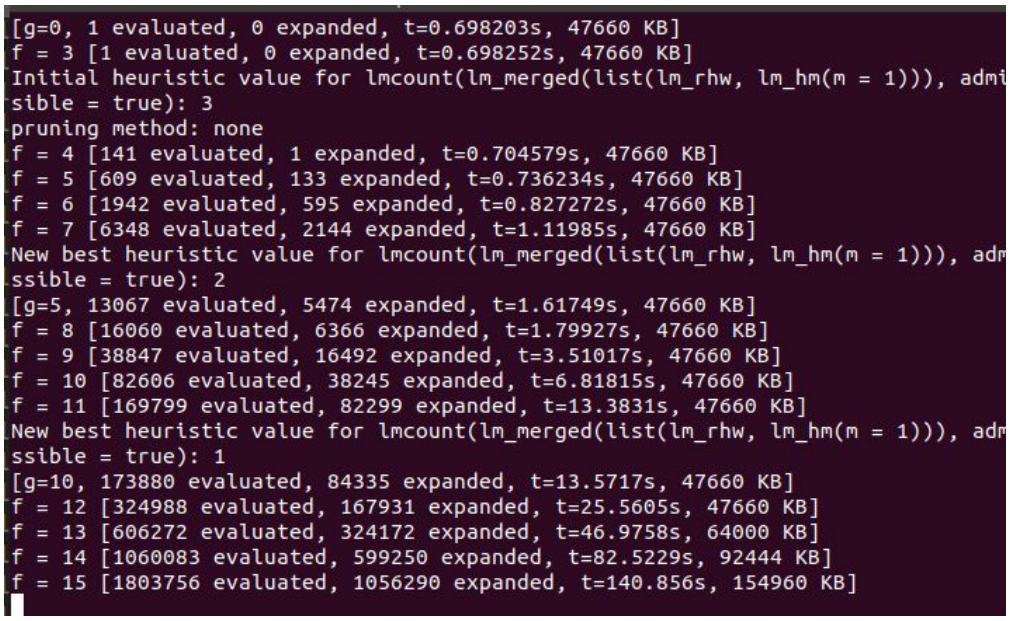
"./fast-downward.py --overall-time-limit 60 --alias lama-first --plan-file myplan.txt Domain Problem"

The Problem.pdll and myplan.txt files carry the numbers of the puzzles. For example, for the puzzle named level1.sok; Problem1.pdll and myplan1.txt files are created.

Since there is a 60 seconds time limit per puzzles for finding a solution, not every puzzle had the myplan.txt file. Under 1 minute with the LAM A solver which uses greedy best first search algorithm which is not an optimal algorithm, we found satisfying solutions for the levels 1,2,8,9,11,13,14,15,16,20,22 and 24.

Next, we tried our code with A* search to see if the solver could find an optimal plan for the puzzles we found solution with the greedy search. For this, we are automatically running the code with the below line inside of sokoban.py file.

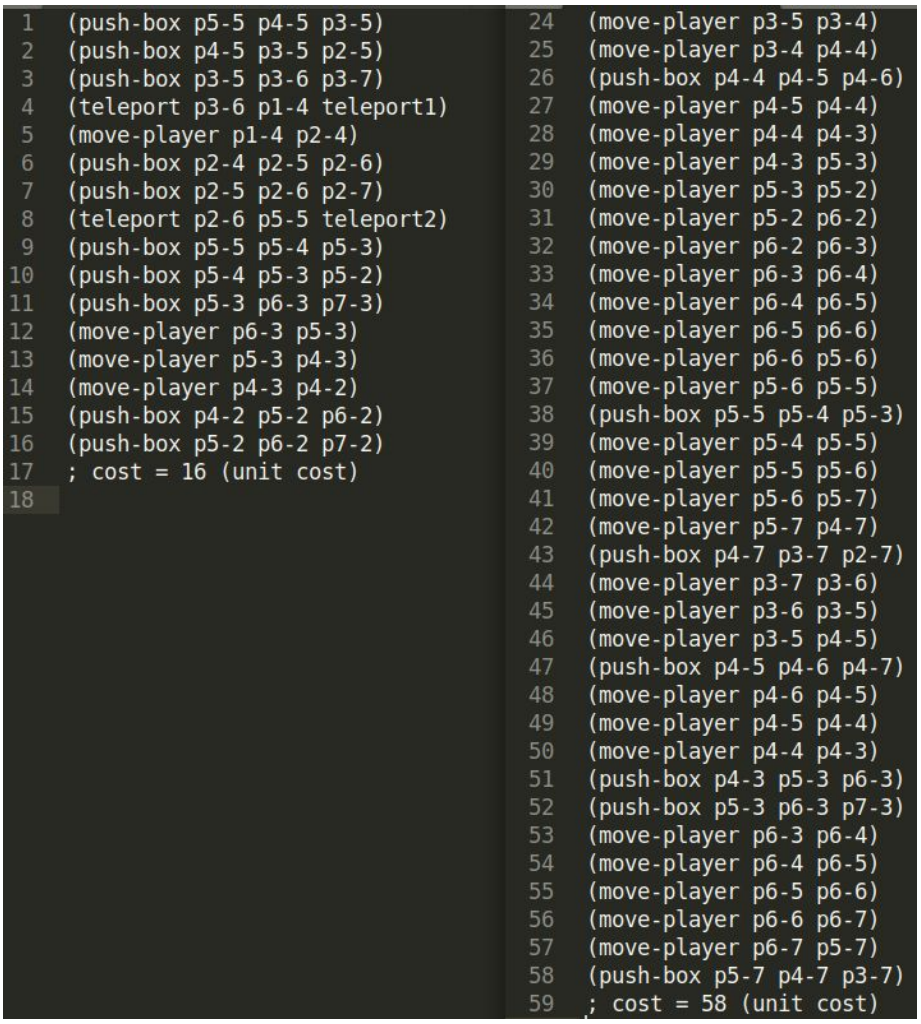
"./fast-downward.py --overall-time-limit 60 --alias seq-opt-bjolp --plan-file myplan.txt Domain Problem"



Running example shown for level1.sok

Only for level2.sok puzzle A* solver found a solution under 1 minute. We found that A* takes more time to solve than greedy best first search.

- A comparison between an optimal solution using A* and admissible and non optimal solution using Greedy Best First Search for level2.sok:



```
(push-box p5-5 p4-5 p3-5)
(push-box p4-5 p3-5 p2-5)
(push-box p3-5 p3-6 p3-7)
(teleport p3-6 p1-4 teleport1)
(move-player p1-4 p2-4)
(push-box p2-4 p2-5 p2-6)
(push-box p2-5 p2-6 p2-7)
(teleport p2-6 p5-5 teleport2)
(push-box p5-5 p5-4 p5-3)
(push-box p5-4 p5-3 p5-2)
(push-box p5-3 p6-3 p7-3)
(move-player p6-3 p5-3)
(move-player p5-3 p4-3)
(move-player p4-3 p4-2)
(push-box p4-2 p5-2 p6-2)
(push-box p5-2 p6-2 p7-2)
; cost = 16 (unit cost)
```

```
(move-player p3-5 p3-4)
(move-player p3-4 p4-4)
(push-box p4-4 p4-5 p4-6)
(move-player p4-5 p4-4)
(move-player p4-4 p4-3)
(move-player p4-3 p5-3)
(move-player p5-3 p5-2)
(move-player p5-2 p6-2)
(move-player p6-2 p6-3)
(move-player p6-3 p6-4)
(move-player p6-4 p6-5)
(move-player p6-5 p6-6)
(move-player p6-6 p5-6)
(move-player p5-6 p5-5)
(push-box p5-5 p5-4 p5-3)
(move-player p5-4 p5-5)
(move-player p5-5 p5-6)
(move-player p5-6 p5-7)
(move-player p5-7 p4-7)
(push-box p4-7 p3-7 p2-7)
(move-player p3-7 p3-6)
(move-player p3-6 p3-5)
(move-player p3-5 p4-5)
(push-box p4-5 p4-6 p4-7)
(move-player p4-6 p4-5)
(move-player p4-5 p4-4)
(move-player p4-4 p4-3)
(push-box p4-3 p5-3 p6-3)
(push-box p5-3 p6-3 p7-3)
(move-player p6-3 p6-4)
(move-player p6-4 p6-5)
(move-player p6-5 p6-6)
(move-player p6-6 p6-7)
(move-player p6-7 p5-7)
(push-box p5-7 p4-7 p3-7)
; cost = 58 (unit cost)
```

Optimal Solution(A)* *Satisfying Solution(Greedy BFS)*

- As we see from the image above, A* returns with fewer cost then Greedy BFS.