HW3

October 6, 2014

1 Homework 1: Classical Planning

Robot Intelligence - Planning CS 4649/7649, Fall 2014
Instructor: Sungmoon Joo
9/29/14 ## Team 3 * Siddharth Choudhary * Varun Murali * Yosef Razin * Ruffin White

1.1 1) Towers of Hanoi

A famous problem in classical planning is the Towers of Hanoi. Apparently, some priests in Vietnam are required to stack enormous discs from one tower to another by command of an ancient prophecy. Lets help them out with modern automation. The discs must always be stacked in order of increasing height. The goal is to move all the discs from the first tower to the third. Experiment with at least two different classical planners to solve this problem. Links to the domain are provided on the course web page. The page also contains links to some recommended planners. You are welcome and recommended to try other planners as well.

```
In [26]: from IPython.display import Image
         from IPython.display import SVG
         from IPython.display import FileLink, FileLinks
```

```
In [2]: Image("Figures/Towers of Hanoi.png")
```

Out[2]:

We'll specify the relative path to each of the planners...

```
In [3]: planner_path = "../../../Documents/Code/Planners/"
        blackbox   = planner_path + "Blackbox43LinuxBinary/blackbox"
        satplan    = planner_path + "SatPlan2006_LinuxBin/satplan"
        vhpop      = planner_path + "vhpop-2.2.1/vhpop"
        graphplan  = planner_path + "Graphplan/graphplan"
        pyperplan  = planner_path + "pyperplan/src/pyperplan.py"
        # shop       = "../../../Documents/Code/Planners/shop2-2.9.0/"
```

.. and then the relative path of the domain and problem .pddl files

```
In [4]: tower_path = "/home/tox/git/HW1_Team3/Code/Towers/"
        domain    = tower_path + "hanoi-domain.pddl"
        problem   = tower_path + "hanoi2.pddl"
```

Let's go ahead and try out the blackbox planner

```
In [8]: %%bash -s "$blackbox" "$domain" "$problem"
        $1 -o $2 -f $3 -solver graphplan
```

```
blackbox version 43
command line:  ../../../Documents/Code/Planners/Blackbox43LinuxBinary/blackbox -o /home/tox/git/HW1_Team

Begin solver specification
    -maxint        0   -maxsec 0.000000  graphplan
End solver specification
Loading domain file: /home/tox/git/HW1_Team3/Code/Towers/hanoi-domain.pddl
Loading fact file: /home/tox/git/HW1_Team3/Code/Towers/hanoi2.pddl
Problem name: hanoi-3
Facts loaded.
time: 1, 24 facts and 6 exclusive pairs.
time: 2, 27 facts and 17 exclusive pairs.
time: 3, 31 facts and 35 exclusive pairs.
Goals reachable at 3 steps but mutually exclusive.
time: 4, 33 facts and 42 exclusive pairs.
Goals first reachable in 4 steps.
297 nodes created.

#####################################################
goals at time 5:
  on_d1_d2 on_d2_d3 on_d3_p1


-----------------------------------------------------
Invoking solver graphplan
Result is Unsat
Iteration was 2

-----------------------------------------------------
Can't solve in 4 steps
time: 5, 33 facts and 41 exclusive pairs.
104 new nodes added.

#####################################################
goals at time 6:
  on_d1_d2 on_d2_d3 on_d3_p1


-----------------------------------------------------
Invoking solver graphplan
Result is Unsat
Iteration was 20

-----------------------------------------------------
```

```
Can't solve in 5 steps
time: 6, 33 facts and 41 exclusive pairs.
104 new nodes added.

#####################################################
goals at time 7:
  on_d1_d2 on_d2_d3 on_d3_p1


-----------------------------------------------------
Invoking solver graphplan
Result is Unsat
Iteration was 70

-----------------------------------------------------
Can't solve in 6 steps
time: 7, 33 facts and 41 exclusive pairs.
104 new nodes added.

#####################################################
goals at time 8:
  on_d1_d2 on_d2_d3 on_d3_p1


-----------------------------------------------------
Invoking solver graphplan
Result is Sat
Iteration was 239
Performing plan justification:
   0 actions were pruned in 0.00 seconds

-----------------------------------------------------
Begin plan
1 (move-disk d1 d2 p1)
2 (move-disk d2 d3 p2)
3 (move-disk d1 p1 d2)
4 (move-disk d3 p3 p1)
5 (move-disk d1 d2 p3)
6 (move-disk d2 p2 d3)
7 (move-disk d1 p3 d2)
End plan
-----------------------------------------------------

7 total actions in plan
25 entries in hash table, 13 hash hits, avg set size 8
44 total set-creation steps (entries + hits + plan length - 1)
38 actions tried

#####################################################
Total elapsed time:   0.00 seconds
Time in milliseconds: 3

#####################################################
```

Ok, so that worked out. Now lets try VHPOP.

```
In [9]: %%bash -s "$vhpop" "$domain" "$problem"
        $1  -f LCFR -l 10000 -f MW -l unlimited $2 $3
```

```
;hanoi-3
1:(move-disk d1 d2 p1)
2:(move-disk d2 d3 p2)
3:(move-disk d1 p1 d2)
4:(move-disk d3 p3 p1)
5:(move-disk d1 d2 p3)
6:(move-disk d2 p2 d3)
7:(move-disk d1 p3 d2)
Time: 696
```

1.1.1 a) Explain the method by which each of the two planners finds a solution.

The first planner applied, Blackbox, is a planning system that combines satplan and graphplan. Basically, the planner parses the PDDL files specified with STRIPS notation into a Boolean, or propositional, satisfiability problem and then applies several different types of satisfiability engines. The name Blackbox refers to the fact that the plan generator and the SAT solver know nothing about each other, thus permitting a flexible system to try out different engines to use. The particular solver used here is just graphplan with default parameters.

Graphplan uses a planning graph to reduce the search from using just the state space. Instead of node and edges representing possible states and actions respectively as in a state space graph, a plan graph uses atomic facts as nodes and two diffent kind of edges. The first type links atomic facts to actions, for which the atomic is a condition for the action. The second links from actions to atomic facts that it effects by making them true or false. Graphplan then iteratively grows the planning graph by proving that there are no solutions of length l-1, where l is the length of the current plan in question, this is done by backward chaining. When exploring the current plan, graphplan then supposes that the goals are true, and then looks for the actions and previous states from which the goals can be achieved. Paths that prove to be impossible due to incompatibility information from mutex conditions are then pruned.

> "Graph is not the state-space graph, which of course could be huge. In fact, unlike the state-space graph in which a plan is a path through the graph, in a Planning Graph a plan is essentially a flow in the network flow sense." graphplan.pdf

The second planner applied, VHPOP is a partial order causal link (POCL) planner loosely based on UCPOP from University of Washington. Written by Håkan L. S. Younes, VHPOP gained recognition during 3rd International Planning Competition (2002) as Best Newcomer and thus reviving the study partial order planning. VHPOP combines from strategies for POCL planning with other developments in the field of domain independent planning such as distance based heuristics and reachability analysis. VHPOP efficiently implements a set of common flaw selection strategies.

Note that: > "A flaw in POCL planning is either an unlinked precondition (called open condition) for an action, or a threatened causal link. While flaw selection is not a backtracking point in the search through plan space for a complete plan, the order in which flaws are resolved can have a dramatic effect on the number of plans searched before a solution is found. The role of flaw selection in POCL planning is similar to the role of variable selection in constraint programming." jair2003.pdf

1.1.2 b) Which planner was fastest?

The faster planner in this instance is the Blackbox planner with a run time of around 2ms on average, where as VHPOP takes around 700ms on average.

1.1.3 c) Explain why the winning planner might be more effective on this problem.

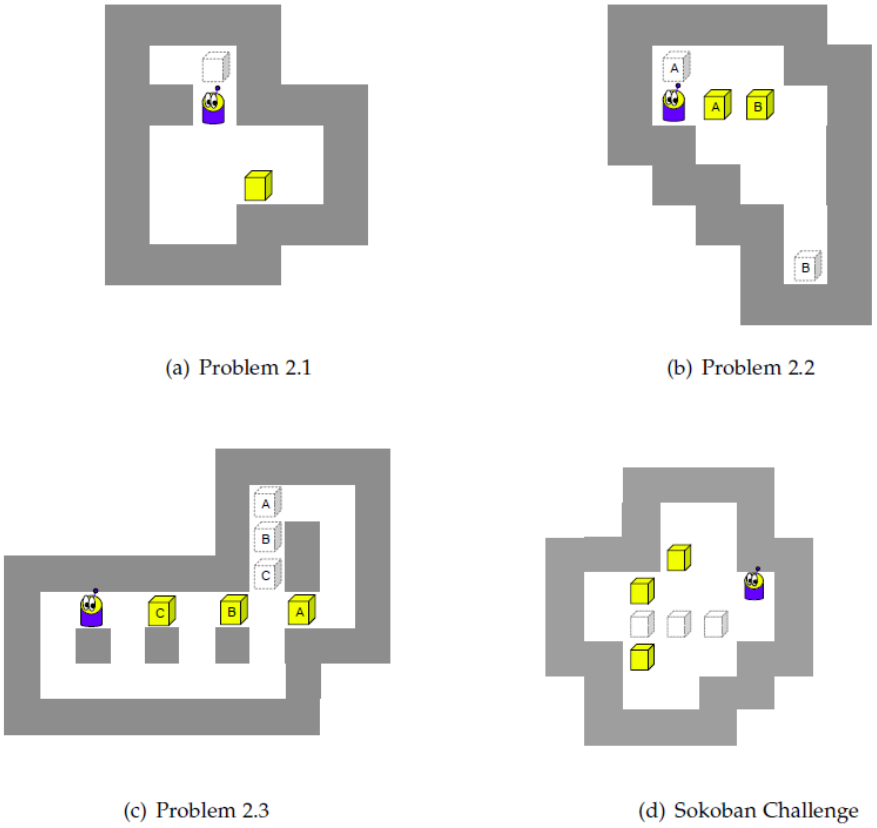
A possible reason for this great disparity might have to do with the overhead that the partial-order planning introduces combined with the brevity of this particular problem results in a meager performance as compared to graphplan that combines aspects of both total-order and partial-order planners.

1.2 2) Sokoban PDDL

During the times of Pong, Pac-Man and Tetris, Hiroyuki Imabayashi created an complex game that tested the human abilities of planning: Sokoban. Many folks are still addicted to solving Sokoban puzzles and you can join them by playing any of the versions freely distributed on the web. The goal is for the human, or robot, to push all the boxes into the desired locations. The robot can move horizontally and vertically and push one box at a time.

```
In [5]: Image("Figures/Sokoban 1.png")
```

Out[5]:



(a) Problem 2.1

(b) Problem 2.2

(c) Problem 2.3

(d) Sokoban Challenge

Describe the Sokoban domain in PDDL. For each of the problems in Figure 2, define the problem in PDDL. You can either use the target lettering given in the picture or let the planner move any box to any target square. For the challenge problem, any box in any location is a solution. In the challenge, PDDL should NOT inform the robot which box should go to which location. In addition you may also try other problems you invent or find on the web. How well do your two planners perform on these problems? If no planner seems to be solving it, perhaps you should consider a different method for defining your problems.

1.2.1 a) Show successful plans from at least one planner on the three Sokoban problems in Figure 2(1-3). The challenge problem is optional.

Ok, so lets import our python function that made in our scratch book to generate our problem PDDL files, this will come in handy as writing and debugging them by hand can be painful.

```
In [6]: %run 'Scratch Book.ipynb'
```

Then let's define the path for the text and pddl files we'll be using. After that, we can save the generated problem PDDL file once we've interpreted the map of the world from the text file.

```
In [7]: world_path = "/home/tox/git/HW1_Team3/Code/Worlds/"

        domain   = world_path + "sokoban_domain.pddl"
        world    = world_path + "world1.txt"
        problem  = world_path + "world1_proble.pddl"
        solution = world_path + "world1_proble.pddl.soln"

        sokoban = Sokoban(world, labeled_boxes = False)
        sokoban.writeProblem(problem)
```

Let's take a look at the domain file first to see what requirements our planners will need fulfill, as well as what predicates exist and what actions can be taken, along with each's parameters, preconditions, and effects.

```
In [10]: %%bash -s "cat" "$domain"
         $1 $2
```

```
; The domain for our sokoban problem is simply named sokoban
(define (domain sokoban)

; Our sokoban domain declares the most basic subset of PDDL, consisting of STRIPS as the only requireme
(:requirements :strips)

; For our predicates, we'll specify the name and number of arguments used for each
(:predicates
        (navigable ?location)                ; Is the location navigatble
        (direction ?direct)                        ; Is the object a direction
        (block-at ?box ?location)        ; Is the box at the given location
        (robot-at ?location)                ; Is the robot at the given location
        (block ?box)                                ; Is the given box a block or crate
        (adjacent ?location1 ?location2 ?direct)         ; Are the two locations adjacent
        (empty ?location)                        ; Is the location empty or avalable
)

; We'll need to be able to move our robot through locations
(:action move

        ; To start, we need a from and to location need to be specified
        :parameters
                (?from ?to ?dir)

        ; Before we effect the current state, we need to make sure
        ; that both the first two given parameters are navigable,
        ; the we have infact a direction
        ; and that given the the direction
```

```
        ; that our to location is both reachable (adjacent) and currently empty
        :precondition
                (and
                        (navigable ?from)
                        (navigable ?to)
                        (direction ?dir)
                        (robot-at ?from)
                        (adjacent ?from ?to ?dir)
                        (empty ?to)
                )

        ; The effect on the current state of the world is that
        ; the location we previosly ocupied will now be vacant with no robot,
        ; the robot now at the "to" location so "to" now ocupied
        :effect
                (and
                        (empty ?from)
                        (robot-at ?to)
                        (not (empty ?to))
                        (not (robot-at ?from))
                )
)

; To menipulate boxes, we'll need to have an action to push them
(:action push

        ; To start, we need locations of the robot, block, destination
        ; plus the corisponding location and block object to be specified
        :parameters
                (?robotLocation ?blockLocation ?freeLocation  ?dirr ?activeBlock)

        ; Before we effect the current state, we need to make sure
        ; that all the first three given parameters are navigable,
        ; and that the block can be pushed into an empty location
        ; that is adjacent in the proper location
        :precondition
                (and
                        (navigable ?robotLocation)
                        (navigable ?blockLocation)
                        (navigable ?freeLocation)
                        (direction ?dirr)
                        (block ?activeBlock)
                        (robot-at ?robotLocation)
                        (block-at ?activeBlock ?blockLocation)
                        (adjacent ?robotLocation ?blockLocation ?dirr)
                        (adjacent ?blockLocation ?freeLocation ?dirr)
                        (empty ?freeLocation)
                )

        ; The effect on the current state of the world is that
        ; the location we previosly ocupied will now be vacant with no robot,
        ; the location the box previosly ocupied will now be ocupied by the robot,
        ; and the free location will now be occupied by the box
        :effect
```

```
                (and
                        (robot-at ?blockLocation)
                        (block-at ?activeBlock ?freeLocation)
                        (empty ?robotLocation)
                        (not (robot-at ?robotLocation))
                        (not (block-at ?activeBlock ?blockLocation))
                        (not (empty ?freeLocation))
                )
)
)
```

Now let us take a look at the first world problem to solve. Here we use _ and X to mark out navigable and non-navigable cells, respectively. The letter o and g represent the block and goal location, respectively, with r representing the robot's starting location. In this manner, we can encode all of the sokoban problems ASCII grid format.

```
In [11]: %%bash -s "cat" "$world"
         $1 $2
```

```
XXXXXX
X_gXXX
XXrXXX
X____X
X__o_X
X__XXX
XXXXXX
```

Let's take a look at the generated output, the problem pddl file we'll have to solve for. We first define the problem and domain names, then list the number of object in the world. After that we then initialize the all of the objects using the predicates available from the domain file. Lastly we then define our goal.

```
In [12]: %%bash -s "cat" "$problem"
         $1 $2
```

```
; XXXXXX
; X_gXXX
; XXrXXX
; X____X
; X__o_X
; X__XXX
; XXXXXX

; This is an auto generated sokoban problem
(define (problem sokoban_problem )

; The domain for our sokoban problem is simply named sokoban
(:domain sokoban)

; Specify list of objects
(:objects

; Specify list of directions
up down left right

; Specify list of blocks
```

```
o

; Specify list of locations
r0-c0 r0-c1 r0-c2 r0-c3 r0-c4 r0-c5
r1-c0 r1-c1 r1-c2 r1-c3 r1-c4 r1-c5
r2-c0 r2-c1 r2-c2 r2-c3 r2-c4 r2-c5
r3-c0 r3-c1 r3-c2 r3-c3 r3-c4 r3-c5
r4-c0 r4-c1 r4-c2 r4-c3 r4-c4 r4-c5
r5-c0 r5-c1 r5-c2 r5-c3 r5-c4 r5-c5
r6-c0 r6-c1 r6-c2 r6-c3 r6-c4 r6-c5

)


; Init direction objects
(:init

; Init direction objects
(direction up)
(direction down)
(direction left)
(direction right)

; Init block objects and locations
(block o)
(block-at o r4-c3)


; Init robot location
(robot-at r2-c2)

; Init navigable objects
(navigable r1-c1)
(navigable r1-c2)
(navigable r2-c2)
(navigable r3-c1)
(navigable r3-c2)
(navigable r3-c3)
(navigable r3-c4)
(navigable r4-c1)
(navigable r4-c2)
(navigable r4-c3)
(navigable r4-c4)
(navigable r5-c1)
(navigable r5-c2)

; Init empty objects
(empty r1-c1)
(empty r1-c2)
(empty r3-c1)
(empty r3-c2)
(empty r3-c3)
(empty r3-c4)
(empty r4-c1)
```

```
(empty r4-c2)
(empty r4-c4)
(empty r5-c1)
(empty r5-c2)

; Init adjacent objects
(adjacent r0-c0 r1-c0 down) (adjacent r0-c0 r0-c1 right)
(adjacent r0-c1 r1-c1 down) (adjacent r0-c1 r0-c0 left) (adjacent r0-c1 r0-c2 right)
(adjacent r0-c2 r1-c2 down) (adjacent r0-c2 r0-c1 left) (adjacent r0-c2 r0-c3 right)
(adjacent r0-c3 r1-c3 down) (adjacent r0-c3 r0-c2 left) (adjacent r0-c3 r0-c4 right)
(adjacent r0-c4 r1-c4 down) (adjacent r0-c4 r0-c3 left) (adjacent r0-c4 r0-c5 right)
(adjacent r0-c5 r1-c5 down) (adjacent r0-c5 r0-c4 left)
(adjacent r1-c0 r0-c0 up) (adjacent r1-c0 r2-c0 down) (adjacent r1-c0 r1-c1 right)
(adjacent r1-c1 r0-c1 up) (adjacent r1-c1 r2-c1 down) (adjacent r1-c1 r1-c0 left) (adjacent r1-c1 r1-c2
(adjacent r1-c2 r0-c2 up) (adjacent r1-c2 r2-c2 down) (adjacent r1-c2 r1-c1 left) (adjacent r1-c2 r1-c3
(adjacent r1-c3 r0-c3 up) (adjacent r1-c3 r2-c3 down) (adjacent r1-c3 r1-c2 left) (adjacent r1-c3 r1-c4
(adjacent r1-c4 r0-c4 up) (adjacent r1-c4 r2-c4 down) (adjacent r1-c4 r1-c3 left) (adjacent r1-c4 r1-c5
(adjacent r1-c5 r0-c5 up) (adjacent r1-c5 r2-c5 down) (adjacent r1-c5 r1-c4 left)
(adjacent r2-c0 r1-c0 up) (adjacent r2-c0 r3-c0 down) (adjacent r2-c0 r2-c1 right)
(adjacent r2-c1 r1-c1 up) (adjacent r2-c1 r3-c1 down) (adjacent r2-c1 r2-c0 left) (adjacent r2-c1 r2-c2
(adjacent r2-c2 r1-c2 up) (adjacent r2-c2 r3-c2 down) (adjacent r2-c2 r2-c1 left) (adjacent r2-c2 r2-c3
(adjacent r2-c3 r1-c3 up) (adjacent r2-c3 r3-c3 down) (adjacent r2-c3 r2-c2 left) (adjacent r2-c3 r2-c4
(adjacent r2-c4 r1-c4 up) (adjacent r2-c4 r3-c4 down) (adjacent r2-c4 r2-c3 left) (adjacent r2-c4 r2-c5
(adjacent r2-c5 r1-c5 up) (adjacent r2-c5 r3-c5 down) (adjacent r2-c5 r2-c4 left)
(adjacent r3-c0 r2-c0 up) (adjacent r3-c0 r4-c0 down) (adjacent r3-c0 r3-c1 right)
(adjacent r3-c1 r2-c1 up) (adjacent r3-c1 r4-c1 down) (adjacent r3-c1 r3-c0 left) (adjacent r3-c1 r3-c2
(adjacent r3-c2 r2-c2 up) (adjacent r3-c2 r4-c2 down) (adjacent r3-c2 r3-c1 left) (adjacent r3-c2 r3-c3
(adjacent r3-c3 r2-c3 up) (adjacent r3-c3 r4-c3 down) (adjacent r3-c3 r3-c2 left) (adjacent r3-c3 r3-c4
(adjacent r3-c4 r2-c4 up) (adjacent r3-c4 r4-c4 down) (adjacent r3-c4 r3-c3 left) (adjacent r3-c4 r3-c5
(adjacent r3-c5 r2-c5 up) (adjacent r3-c5 r4-c5 down) (adjacent r3-c5 r3-c4 left)
(adjacent r4-c0 r3-c0 up) (adjacent r4-c0 r5-c0 down) (adjacent r4-c0 r4-c1 right)
(adjacent r4-c1 r3-c1 up) (adjacent r4-c1 r5-c1 down) (adjacent r4-c1 r4-c0 left) (adjacent r4-c1 r4-c2
(adjacent r4-c2 r3-c2 up) (adjacent r4-c2 r5-c2 down) (adjacent r4-c2 r4-c1 left) (adjacent r4-c2 r4-c3
(adjacent r4-c3 r3-c3 up) (adjacent r4-c3 r5-c3 down) (adjacent r4-c3 r4-c2 left) (adjacent r4-c3 r4-c4
(adjacent r4-c4 r3-c4 up) (adjacent r4-c4 r5-c4 down) (adjacent r4-c4 r4-c3 left) (adjacent r4-c4 r4-c5
(adjacent r4-c5 r3-c5 up) (adjacent r4-c5 r5-c5 down) (adjacent r4-c5 r4-c4 left)
(adjacent r5-c0 r4-c0 up) (adjacent r5-c0 r6-c0 down) (adjacent r5-c0 r5-c1 right)
(adjacent r5-c1 r4-c1 up) (adjacent r5-c1 r6-c1 down) (adjacent r5-c1 r5-c0 left) (adjacent r5-c1 r5-c2
(adjacent r5-c2 r4-c2 up) (adjacent r5-c2 r6-c2 down) (adjacent r5-c2 r5-c1 left) (adjacent r5-c2 r5-c3
(adjacent r5-c3 r4-c3 up) (adjacent r5-c3 r6-c3 down) (adjacent r5-c3 r5-c2 left) (adjacent r5-c3 r5-c4
(adjacent r5-c4 r4-c4 up) (adjacent r5-c4 r6-c4 down) (adjacent r5-c4 r5-c3 left) (adjacent r5-c4 r5-c5
(adjacent r5-c5 r4-c5 up) (adjacent r5-c5 r6-c5 down) (adjacent r5-c5 r5-c4 left)
(adjacent r6-c0 r5-c0 up) (adjacent r6-c0 r6-c1 right)
(adjacent r6-c1 r5-c1 up) (adjacent r6-c1 r6-c0 left) (adjacent r6-c1 r6-c2 right)
(adjacent r6-c2 r5-c2 up) (adjacent r6-c2 r6-c1 left) (adjacent r6-c2 r6-c3 right)
(adjacent r6-c3 r5-c3 up) (adjacent r6-c3 r6-c2 left) (adjacent r6-c3 r6-c4 right)
(adjacent r6-c4 r5-c4 up) (adjacent r6-c4 r6-c3 left) (adjacent r6-c4 r6-c5 right)
(adjacent r6-c5 r5-c5 up) (adjacent r6-c5 r6-c4 left)

; Init Done
)
; Define goal states
(:goal (and (block-at o r1-c2)))
```

```
; Problem Define Done
)
```

Now let's try out our Blackbox planner again and see if our domain and problem files allow for a working answer.

```
In [13]: %%bash -s "$blackbox" "$domain" "$problem"
         $1 -o $2 -f $3
```

```
blackbox version 43
command line:  ../../../Documents/Code/Planners/Blackbox43LinuxBinary/blackbox -o /home/tox/git/HW1_Team

Begin solver specification
    -maxint        0   -maxsec 10.000000  graphplan
    -maxint        0   -maxsec 0.000000  chaff
End solver specification
Loading domain file: /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
Loading fact file: /home/tox/git/HW1_Team3/Code/Worlds/world1_proble.pddl
Problem name: sokoban-problem
Facts loaded.
time: 1, 176 facts and 6 exclusive pairs.
time: 2, 180 facts and 28 exclusive pairs.
time: 3, 185 facts and 79 exclusive pairs.
time: 4, 188 facts and 117 exclusive pairs.
time: 5, 189 facts and 124 exclusive pairs.
time: 6, 190 facts and 130 exclusive pairs.
time: 7, 190 facts and 122 exclusive pairs.
time: 8, 191 facts and 131 exclusive pairs.
time: 9, 193 facts and 162 exclusive pairs.
time: 10, 193 facts and 148 exclusive pairs.
time: 11, 193 facts and 139 exclusive pairs.
time: 12, 194 facts and 153 exclusive pairs.
time: 13, 195 facts and 167 exclusive pairs.
time: 14, 196 facts and 180 exclusive pairs.
Goals first reachable in 14 steps.
5912 nodes created.

#####################################################
goals at time 15:
  block-at_o_r1-c2


-----------------------------------------------------
Invoking solver graphplan
Result is Sat
Iteration was 385
Performing plan justification:
   0 actions were pruned in 0.00 seconds

-----------------------------------------------------
Begin plan
1 (move r2-c2 r3-c2 down)
2 (move r3-c2 r3-c3 right)
3 (move r3-c3 r3-c4 right)
4 (move r3-c4 r4-c4 down)
```

```
5 (push r4-c4 r4-c3 r4-c2 left o)
6 (move r4-c3 r3-c3 up)
7 (move r3-c3 r3-c2 left)
8 (move r3-c2 r3-c1 left)
9 (move r3-c1 r4-c1 down)
10 (move r4-c1 r5-c1 down)
11 (move r5-c1 r5-c2 right)
12 (push r5-c2 r4-c2 r3-c2 up o)
13 (push r4-c2 r3-c2 r2-c2 up o)
14 (push r3-c2 r2-c2 r1-c2 up o)
End plan
----------------------------------------------------

14 total actions in plan
0 entries in hash table,
13 total set-creation steps (entries + hits + plan length - 1)
14 actions tried

#####################################################
Total elapsed time:   0.32 seconds
Time in milliseconds: 322

#####################################################
```

Awesome, we have at least one solution. Let's check if VHPOP well do any better this time.

```
In [14]: %%bash -s "$vhpop" "$domain" "$problem"
         $1  -f LCFR -l 10000 -f MW -l unlimited $2 $3

Process is terminated.
```

Well, VHPOP never seemed to be able to finish this example under 15 min, so lets try something else. Pyperplan is a handy python based planning library. Lets try out a simple breadth first search for this as see how we do.

```
In [15]: %%bash -s "$pyperplan" "$domain" "$problem"
         $1 $2 $3 $4 --plugins visualizer -s bfs

2014-10-05 00:18:53,023 INFO     Using plugin: visualizer
2014-10-05 00:18:53,023 INFO     problem: /home/tox/git/HW1_Team3/Code/Worlds/world1_proble.pddl
2014-10-05 00:18:53,023 INFO     using search: breadth_first_search
2014-10-05 00:18:53,023 INFO     using heuristic: None
2014-10-05 00:18:53,023 INFO     using transition reduction: NoneTransitionPruning
2014-10-05 00:18:53,023 INFO     using state reduction: NoneStatePruning
2014-10-05 00:18:53,036 INFO     Parsing domain /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
domFile: /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
2014-10-05 00:18:53,038 INFO     Parsing problem /home/tox/git/HW1_Team3/Code/Worlds/world1_proble.pddl
2014-10-05 00:18:53,044 INFO     7 Predicates parsed
2014-10-05 00:18:53,045 INFO     2 Actions parsed
2014-10-05 00:18:53,045 INFO     47 Objects parsed
2014-10-05 00:18:53,045 INFO     0 Constants parsed
2014-10-05 00:18:53,045 INFO     Grounding start: sokoban_problem
2014-10-05 00:18:53,148 INFO     Relevance analysis removed 5 facts
2014-10-05 00:18:53,149 INFO     Grounding end: sokoban_problem
2014-10-05 00:18:53,149 INFO     38 Variables created
```

```
2014-10-05 00:18:53,149 INFO     48 Operators created
2014-10-05 00:18:53,149 INFO     Search start: sokoban_problem
2014-10-05 00:18:53,191 INFO     Goal reached. Start extraction of solution.
2014-10-05 00:18:53,191 INFO     95 Nodes expanded
2014-10-05 00:18:53,195 INFO     Search end: sokoban_problem
2014-10-05 00:18:53,195 INFO     Wall-clock search time: 0.04592
2014-10-05 00:18:53,904 INFO     Plan length: 14
2014-10-05 00:18:53,905 INFO     validate could not be found on the PATH so the plan can not be validate
```

Sweet, we got a another solution. Let's check that it matches up with what Blackbox returned.

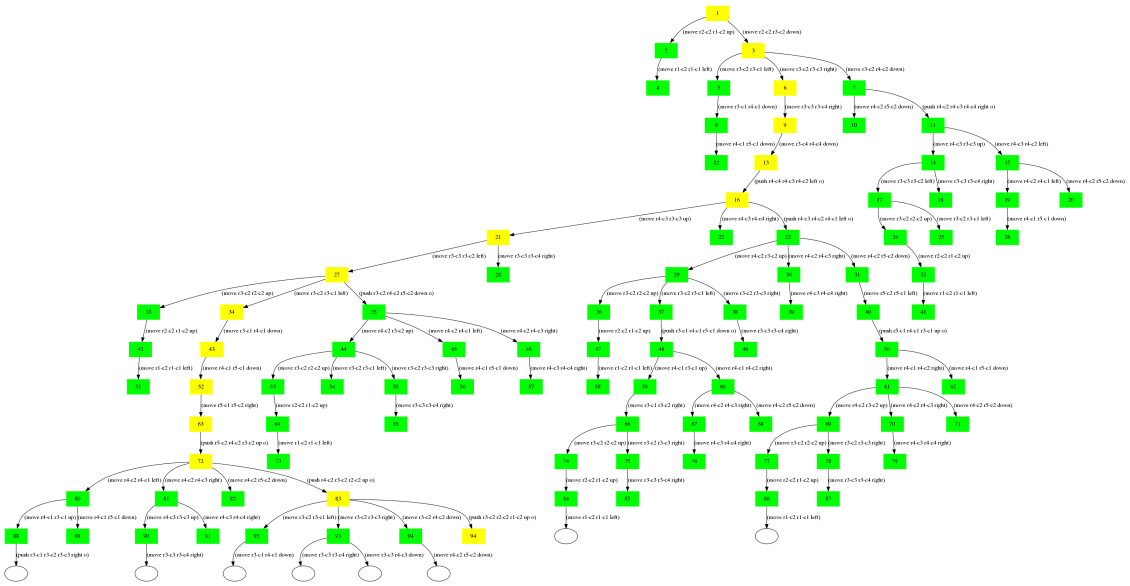
```
In [16]: %%bash -s "cat" "$solution"
         $1 $2
```

```
(move r2-c2 r3-c2 down)
(move r3-c2 r3-c3 right)
(move r3-c3 r3-c4 right)
(move r3-c4 r4-c4 down)
(push r4-c4 r4-c3 r4-c2 left o)
(move r4-c3 r3-c3 up)
(move r3-c3 r3-c2 left)
(move r3-c2 r3-c1 left)
(move r3-c1 r4-c1 down)
(move r4-c1 r5-c1 down)
(move r5-c1 r5-c2 right)
(push r5-c2 r4-c2 r3-c2 up o)
(push r4-c2 r3-c2 r2-c2 up o)
(push r3-c2 r2-c2 r1-c2 up o)
```

Sure enough, they are the same, and just about the same as I would do in maintaining minimal movment to maximize efficiency. Let's take a look at the explored region of the state space using a breadth first search.

```
In [17]: Image("Worlds-bfs-hff-world1_proble.png")
```

Out[17]:



As you can see, our goal happens to be the last explored stated on the deepest currently explored level. The green colors the explored nodes in the graph, the empty circles are the yet to be explored nodes, and the yellow path depicts the optimal path to the goal. We know this to be optimal as with bread first search, if a closer goal where to be found, we would have explored it when searching previous breadths of the explored tree above. We also know BFS to be complete, as if a solution exists, we will eventually find it, given we have a enough memory to handle worst case space complexity of $O(b^d)$, where b – maximum branching factor of the search tree, and d – depth of the least-cost solution to the goal. Note: in the nbviewer, you can right click the image and open it in a new tab to view the original size. The other images may be downscaled for size limitations.

```
In [21]: domain   = world_path + "sokoban_domain.pddl"
         world    = world_path + "world2.txt"
         problem  = world_path + "world2_proble.pddl"
         solution = world_path + "world2_proble.pddl.soln"

         sokoban = Sokoban(world, labeled_boxes = False)
         sokoban.writeProblem(problem)
```

Now let's try solving the second sokoban problem. Doing this myself was a bit tricky and didn't quite see the solution at first as one had to rearrange the blocks in a tight configuration several times to get to a suitable position to push the blocks to the goal. This turned out to be to challenging for blackbox and graphplan with default configs, so let us move on using pyperplan instead.

```
In [22]: %%bash -s "$pyperplan" "$domain" "$problem"
         $1 $2 $3 $4 --plugins visualizer

2014-10-05 18:55:51,662 INFO     Using plugin: visualizer
2014-10-05 18:55:51,663 INFO     problem: /home/tox/git/HW1_Team3/Code/Worlds/world2_proble.pddl
2014-10-05 18:55:51,663 INFO     using search: breadth_first_search
2014-10-05 18:55:51,663 INFO     using heuristic: None
2014-10-05 18:55:51,663 INFO     using transition reduction: NoneTransitionPruning
2014-10-05 18:55:51,663 INFO     using state reduction: NoneStatePruning
2014-10-05 18:55:51,663 INFO     Parsing domain /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
domFile: /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
2014-10-05 18:55:51,665 INFO     Parsing problem /home/tox/git/HW1_Team3/Code/Worlds/world2_proble.pddl
2014-10-05 18:55:51,673 INFO     7 Predicates parsed
2014-10-05 18:55:51,673 INFO     2 Actions parsed
2014-10-05 18:55:51,673 INFO     54 Objects parsed
2014-10-05 18:55:51,673 INFO     0 Constants parsed
2014-10-05 18:55:51,674 INFO     Grounding start: sokoban_problem
2014-10-05 18:55:51,898 INFO     Relevance analysis removed 12 facts
2014-10-05 18:55:51,898 INFO     Grounding end: sokoban_problem
2014-10-05 18:55:51,898 INFO     56 Variables created
2014-10-05 18:55:51,898 INFO     76 Operators created
2014-10-05 18:55:51,898 INFO     Search start: sokoban_problem
2014-10-05 18:55:52,348 INFO     Goal reached. Start extraction of solution.
2014-10-05 18:55:52,348 INFO     953 Nodes expanded
2014-10-05 18:55:52,356 INFO     Search end: sokoban_problem
2014-10-05 18:55:52,356 INFO     Wall-clock search time: 0.45802
2014-10-05 18:55:57,447 INFO     Plan length: 32
2014-10-05 18:55:57,449 INFO     validate could not be found on the PATH so the plan can not be validat
```

OK, let's take a look at the optimal number of moves required when using BFS.

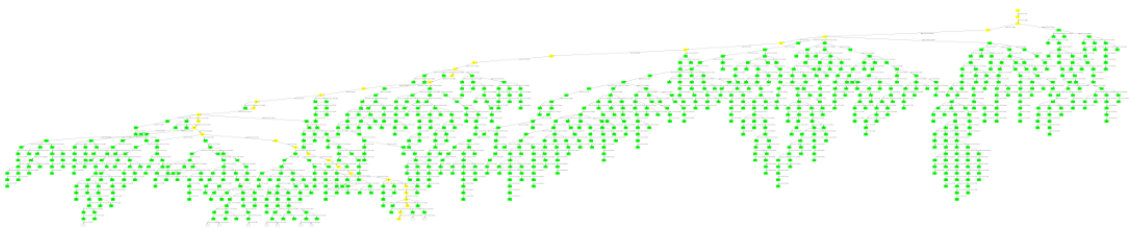
```
In [23]: %%bash -s "cat" "$solution"
         $1 $2
```

```
(move r2-c1 r1-c1 up)
(move r1-c1 r1-c2 right)
(move r1-c2 r1-c3 right)
(push r1-c3 r2-c3 r3-c3 down b)
(move r2-c3 r1-c3 up)
(move r1-c3 r1-c2 left)
(move r1-c2 r1-c1 left)
(move r1-c1 r2-c1 down)
(push r2-c1 r2-c2 r2-c3 right a)
(move r2-c2 r3-c2 down)
(push r3-c2 r3-c3 r3-c4 right b)
(move r3-c3 r3-c2 left)
(move r3-c2 r2-c2 up)
(move r2-c2 r1-c2 up)
(move r1-c2 r1-c3 right)
(push r1-c3 r2-c3 r3-c3 down a)
(move r2-c3 r2-c4 right)
(push r2-c4 r3-c4 r4-c4 down b)
(push r3-c4 r4-c4 r5-c4 down b)
(push r4-c4 r5-c4 r6-c4 down b)
(move r5-c4 r4-c4 up)
(move r4-c4 r4-c3 left)
(push r4-c3 r3-c3 r2-c3 up a)
(move r3-c3 r3-c4 right)
(move r3-c4 r2-c4 up)
(push r2-c4 r2-c3 r2-c2 left a)
(move r2-c3 r3-c3 down)
(move r3-c3 r3-c2 left)
(push r3-c2 r2-c2 r1-c2 up a)
(move r2-c2 r2-c3 right)
(move r2-c3 r1-c3 up)
(push r1-c3 r1-c2 r1-c1 left a)
```

We need 32 moves in total; as we can see things are now getting a bit complex for such a small map. The multiple number of blocks adds to the complexity here. Look at the explored state space used to find the solution. We'll see our path to the goal transversing from the start state, down to the last explored node at the deepest explored depth.

```
In [24]: Image("Figures/Worlds-bfs-hff-world2_proble.png", width=1080)
```

Out[24]:



As we can see, it doesn't seem the BFS is scaling nicely. Let's try switching to a diffrent search method. In particular, let's try using a greedy best first search along with a Hybrid Factored Frontier (HFF) algorithm to serve as the heurestic. HFF is sometimes called the relaxed plan h, it is not admissible, but it is informative for our GBF search. Besacue GBF is not optimal, and our heurestic here is not admissible, we may only find a sub-optimal solution.

```
In [25]: %%bash -s "$pyperplan" "$domain" "$problem"
         $1 $2 $3 $4 --plugins visualizer -H hff -s gbf
```

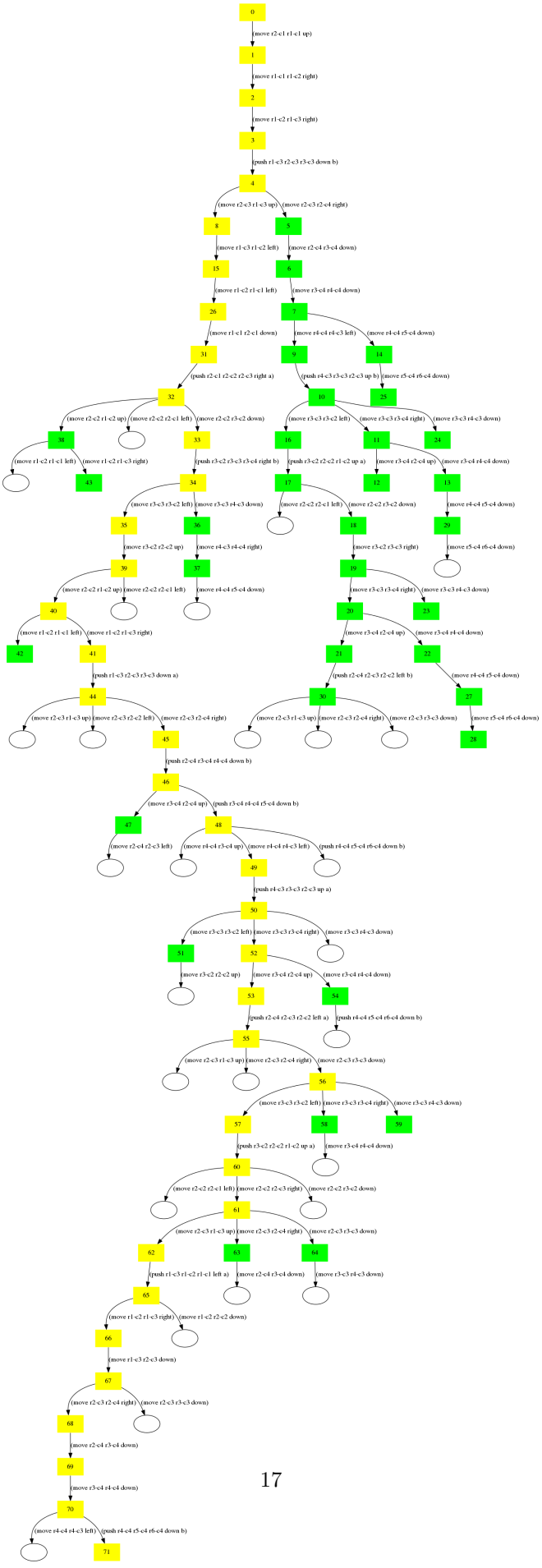
```
2014-10-05 18:55:59,591 INFO     Using plugin: visualizer
2014-10-05 18:55:59,591 INFO     problem: /home/tox/git/HW1_Team3/Code/Worlds/world2_proble.pddl
2014-10-05 18:55:59,591 INFO     using search: greedy_best_first_search
2014-10-05 18:55:59,591 INFO     using heuristic: hFFHeuristic
2014-10-05 18:55:59,591 INFO     using transition reduction: NoneTransitionPruning
2014-10-05 18:55:59,591 INFO     using state reduction: NoneStatePruning
2014-10-05 18:55:59,591 INFO     Parsing domain /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
domFile: /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
2014-10-05 18:55:59,593 INFO     Parsing problem /home/tox/git/HW1_Team3/Code/Worlds/world2_proble.pddl
2014-10-05 18:55:59,602 INFO     7 Predicates parsed
2014-10-05 18:55:59,602 INFO     2 Actions parsed
2014-10-05 18:55:59,602 INFO     54 Objects parsed
2014-10-05 18:55:59,602 INFO     0 Constants parsed
2014-10-05 18:55:59,602 INFO     Grounding start: sokoban_problem
2014-10-05 18:55:59,830 INFO     Relevance analysis removed 12 facts
2014-10-05 18:55:59,830 INFO     Grounding end: sokoban_problem
2014-10-05 18:55:59,830 INFO     56 Variables created
2014-10-05 18:55:59,830 INFO     76 Operators created
2014-10-05 18:55:59,830 INFO     Search start: sokoban_problem
2014-10-05 18:55:59,831 INFO     Initial h value: 15.000000
2014-10-05 18:55:59,914 INFO     Goal reached. Start extraction of solution.
2014-10-05 18:55:59,914 INFO     72 Nodes expanded
2014-10-05 18:55:59,922 INFO     Search end: sokoban_problem
2014-10-05 18:55:59,922 INFO     Wall-clock search time: 0.09148
2014-10-05 18:56:00,529 INFO     Plan length: 36
2014-10-05 18:56:00,530 INFO     validate could not be found on the PATH so the plan can not be validat
```

Alright, we have a new plan. This GBF method found a path to goal taking 36 moves in about 0.091 sec by expanding 72 nodes v.s. what BFS found, a path to goal taking 32 moves in about 0.458 sec by expanding 953 nodes. So our new solution takes about 4 moves extra, but only took less than one quarter of the time. So based on what may be important to you, time that is takes to make plan, or the time it takes to make a move, the optimality of the solution may be specific to the application.

Lets take a look at the explored tree of the state space for GBF with HFF for comparison.

```
In [26]: Image("Worlds-gbf-hff-world2_proble.png")
```

Out[26]:


0
(move r2-c1 r1-c1 up)
1
(move r1-c1 r1-c2 right)
2
(move r1-c2 r1-c3 right)
3
(push r1-c3 r2-c3 r3-c3 down b)
4
(move r2-c3 r1-c3 up)
(move r2-c3 r2-c4 right)
8
5
(move r1-c3 r1-c2 left)
(move r2-c4 r3-c4 down)
15
6
(move r1-c2 r1-c1 left)
(move r3-c4 r4-c4 down)
26
7
(move r1-c1 r2-c1 down)
(move r4-c4 r4-c3 left)
(move r4-c4 r5-c4 down)
31
9
14
(push r2-c1 r2-c2 r2-c3 right a)
(push r4-c3 r3-c3 r2-c3 up b)
(move r5-c4 r6-c4 down)
32
10
25
(move r2-c2 r1-c2 up)
(move r2-c2 r2-c1 left)
(move r2-c2 r3-c2 down)
(move r3-c3 r3-c2 left)
(move r3-c3 r3-c4 right)
(move r3-c3 r4-c3 down)
38
33
16
11
24
(move r1-c2 r1-c1 left)
(move r1-c2 r1-c3 right)
(push r3-c2 r3-c3 r3-c4 right b)
(push r3-c2 r2-c2 r1-c2 up a)
(move r3-c4 r2-c4 up)
(move r3-c4 r4-c4 down)
43
34
17
12
13
(move r3-c3 r3-c2 left)
(move r3-c3 r4-c3 down)
(move r2-c2 r2-c1 left)
(move r2-c2 r3-c2 down)
(move r4-c4 r5-c4 down)
35
36
18
29
(move r3-c2 r2-c2 up)
(move r4-c3 r4-c4 right)
(move r3-c2 r3-c3 right)
(move r5-c4 r6-c4 down)
39
37
19
(move r2-c2 r1-c2 up)
(move r2-c2 r2-c1 left)
(move r4-c4 r5-c4 down)
(move r3-c3 r3-c4 right)
(move r3-c3 r4-c3 down)
40
20
23
(move r1-c2 r1-c1 left)
(move r1-c2 r1-c3 right)
(move r3-c4 r2-c4 up)
(move r3-c4 r4-c4 down)
42
41
21
22
(push r1-c3 r2-c3 r3-c3 down a)
(push r2-c4 r2-c3 r2-c2 left b)
(move r4-c4 r5-c4 down)
44
30
27
(move r2-c3 r1-c3 up)
(move r2-c3 r2-c2 left)
(move r2-c3 r2-c4 right)
(move r2-c3 r1-c3 up)
(move r2-c3 r2-c4 right)
(move r2-c3 r3-c3 down)
(move r5-c4 r6-c4 down)
45
28
(push r2-c4 r3-c4 r4-c4 down b)
46
(move r3-c4 r2-c4 up)
(push r3-c4 r4-c4 r5-c4 down b)
47
48
(move r2-c4 r2-c3 left)
(move r4-c4 r3-c4 up)
(move r4-c4 r4-c3 left)
(push r4-c4 r5-c4 r6-c4 down b)
49
(push r4-c3 r3-c3 r2-c3 up a)
50
(move r3-c3 r3-c2 left)
(move r3-c3 r3-c4 right)
(move r3-c3 r4-c3 down)
51
52
(move r3-c2 r2-c2 up)
(move r3-c4 r2-c4 up)
(move r3-c4 r4-c4 down)
53
54
(push r2-c4 r2-c3 r2-c2 left a)
(push r4-c4 r5-c4 r6-c4 down b)
55
(move r2-c3 r1-c3 up)
(move r2-c3 r2-c4 right)
(move r2-c3 r3-c3 down)
56
(move r3-c3 r3-c2 left)
(move r3-c3 r3-c4 right)
(move r3-c3 r4-c3 down)
57
58
59
(push r3-c2 r2-c2 r1-c2 up a)
(move r3-c4 r4-c4 down)
60
(move r2-c2 r2-c1 left)
(move r2-c2 r2-c3 right)
(move r2-c2 r3-c2 down)
61
(move r2-c3 r1-c3 up)
(move r2-c3 r2-c4 right)
(move r2-c3 r3-c3 down)
62
63
64
(push r1-c3 r1-c2 r1-c1 left a)
(move r2-c4 r3-c4 down)
(move r3-c3 r4-c3 down)
65
(move r1-c2 r1-c3 right)
(move r1-c2 r2-c2 down)
66
(move r1-c3 r2-c3 down)
67
(move r2-c3 r2-c4 right)
(move r2-c3 r3-c3 down)
68
(move r2-c4 r3-c4 down)
69
(move r3-c4 r4-c4 down)
70
(move r4-c4 r4-c3 left)
(push r4-c4 r5-c4 r6-c4 down b)
71

Now let's do the same comparison for the third problem

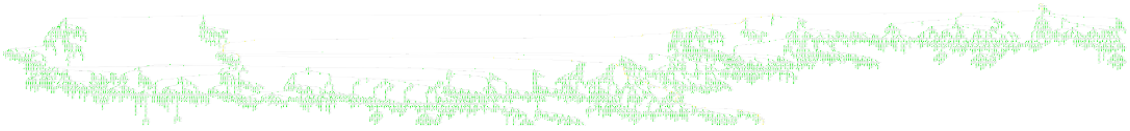
```
In [24]: domain   = world_path + "sokoban_domain.pddl"
         world    = world_path + "world3.txt"
         problem  = world_path + "world3_proble.pddl"
         solution = world_path + "world3_proble.pddl.soln"

         sokoban = Sokoban(world, labeled_boxes = False)
         sokoban.writeProblem(problem)

In [25]: %%bash -s "$pyperplan" "$domain" "$problem"
         $1 $2 $3 $4 --plugins visualizer

2014-10-05 00:19:04,184 INFO     Using plugin: visualizer
2014-10-05 00:19:04,184 INFO     problem: /home/tox/git/HW1_Team3/Code/Worlds/world3_proble.pddl
2014-10-05 00:19:04,184 INFO     using search: breadth_first_search
2014-10-05 00:19:04,184 INFO     using heuristic: None
2014-10-05 00:19:04,184 INFO     using transition reduction: NoneTransitionPruning
2014-10-05 00:19:04,184 INFO     using state reduction: NoneStatePruning
2014-10-05 00:19:04,185 INFO     Parsing domain /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
domFile: /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
2014-10-05 00:19:04,187 INFO     Parsing problem /home/tox/git/HW1_Team3/Code/Worlds/world3_proble.pddl
2014-10-05 00:19:04,204 INFO     7 Predicates parsed
2014-10-05 00:19:04,204 INFO     2 Actions parsed
2014-10-05 00:19:04,204 INFO     95 Objects parsed
2014-10-05 00:19:04,204 INFO     0 Constants parsed
2014-10-05 00:19:04,204 INFO     Grounding start: sokoban_problem
2014-10-05 00:19:06,341 INFO     Relevance analysis removed 17 facts
2014-10-05 00:19:06,341 INFO     Grounding end: sokoban_problem
2014-10-05 00:19:06,341 INFO     135 Variables created
2014-10-05 00:19:06,341 INFO     192 Operators created
2014-10-05 00:19:06,341 INFO     Search start: sokoban_problem
2014-10-05 00:19:11,207 INFO     Goal reached. Start extraction of solution.
2014-10-05 00:19:11,207 INFO     7867 Nodes expanded
2014-10-05 00:19:11,233 INFO     Search end: sokoban_problem
2014-10-05 00:19:11,233 INFO     Wall-clock search time: 4.89248
dot: graph is too large for cairo-renderer bitmaps. Scaling by 0.335329 to fit

2014-10-05 00:20:17,593 INFO     Plan length: 89
2014-10-05 00:20:17,599 INFO     validate could not be found on the PATH so the plan can not be validat

In [26]: %%bash -s "cat" "$solution"
         $1 $2

(move r4-c2 r4-c3 right)
(move r4-c3 r5-c3 down)
(move r5-c3 r6-c3 down)
(move r6-c3 r6-c4 right)
(move r6-c4 r6-c5 right)
(move r6-c5 r5-c5 up)
(move r5-c5 r4-c5 up)
(push r4-c5 r4-c4 r4-c3 left c)
(push r4-c4 r4-c3 r4-c2 left c)
```

```
(move r4-c3 r4-c4 right)
(move r4-c4 r4-c5 right)
(move r4-c5 r5-c5 down)
(move r5-c5 r6-c5 down)
(move r6-c5 r6-c6 right)
(move r6-c6 r6-c7 right)
(move r6-c7 r5-c7 up)
(move r5-c7 r4-c7 up)
(move r4-c7 r3-c7 up)
(move r3-c7 r2-c7 up)
(move r2-c7 r1-c7 up)
(move r1-c7 r1-c8 right)
(move r1-c8 r1-c9 right)
(move r1-c9 r2-c9 down)
(move r2-c9 r3-c9 down)
(move r3-c9 r4-c9 down)
(push r4-c9 r4-c8 r4-c7 left a)
(move r4-c8 r4-c9 right)
(move r4-c9 r3-c9 up)
(move r3-c9 r2-c9 up)
(move r2-c9 r1-c9 up)
(move r1-c9 r1-c8 left)
(move r1-c8 r1-c7 left)
(move r1-c7 r2-c7 down)
(move r2-c7 r3-c7 down)
(push r3-c7 r4-c7 r5-c7 down a)
(push r4-c7 r4-c6 r4-c5 left b)
(push r4-c6 r4-c5 r4-c4 left b)
(move r4-c5 r5-c5 down)
(move r5-c5 r6-c5 down)
(move r6-c5 r6-c6 right)
(move r6-c6 r6-c7 right)
(push r6-c7 r5-c7 r4-c7 up a)
(push r5-c7 r4-c7 r3-c7 up a)
(push r4-c7 r3-c7 r2-c7 up a)
(push r3-c7 r2-c7 r1-c7 up a)
(move r2-c7 r3-c7 down)
(move r3-c7 r4-c7 down)
(move r4-c7 r4-c6 left)
(move r4-c6 r4-c5 left)
(move r4-c5 r5-c5 down)
(move r5-c5 r6-c5 down)
(move r6-c5 r6-c4 left)
(move r6-c4 r6-c3 left)
(move r6-c3 r5-c3 up)
(move r5-c3 r4-c3 up)
(push r4-c3 r4-c4 r4-c5 right b)
(push r4-c4 r4-c5 r4-c6 right b)
(push r4-c5 r4-c6 r4-c7 right b)
(move r4-c6 r4-c5 left)
(move r4-c5 r5-c5 down)
(move r5-c5 r6-c5 down)
(move r6-c5 r6-c6 right)
(move r6-c6 r6-c7 right)
```

```
(move r6-c7 r5-c7 up)
(push r5-c7 r4-c7 r3-c7 up b)
(push r4-c7 r3-c7 r2-c7 up b)
(move r3-c7 r4-c7 down)
(move r4-c7 r4-c6 left)
(move r4-c6 r4-c5 left)
(move r4-c5 r4-c4 left)
(move r4-c4 r4-c3 left)
(move r4-c3 r5-c3 down)
(move r5-c3 r6-c3 down)
(move r6-c3 r6-c2 left)
(move r6-c2 r6-c1 left)
(move r6-c1 r5-c1 up)
(move r5-c1 r4-c1 up)
(push r4-c1 r4-c2 r4-c3 right c)
(push r4-c2 r4-c3 r4-c4 right c)
(push r4-c3 r4-c4 r4-c5 right c)
(push r4-c4 r4-c5 r4-c6 right c)
(push r4-c5 r4-c6 r4-c7 right c)
(move r4-c6 r4-c5 left)
(move r4-c5 r5-c5 down)
(move r5-c5 r6-c5 down)
(move r6-c5 r6-c6 right)
(move r6-c6 r6-c7 right)
(move r6-c7 r5-c7 up)
(push r5-c7 r4-c7 r3-c7 up c)
```

In [3]:
```
Image("Figures/Worlds-bfs-hff-world3_proble.png", width=1080)
```

Out[3]:



Ok, we've found the plan length takes 89 moves and took 4.89 sec and 7,867 nodes to expand to find with breadth first search. Now let's try it with our other method.

In [28]:
```
%%bash -s "$pyperplan" "$domain" "$problem"
$1 $2 $3 $4 --plugins visualizer -H hff -s gbf
```

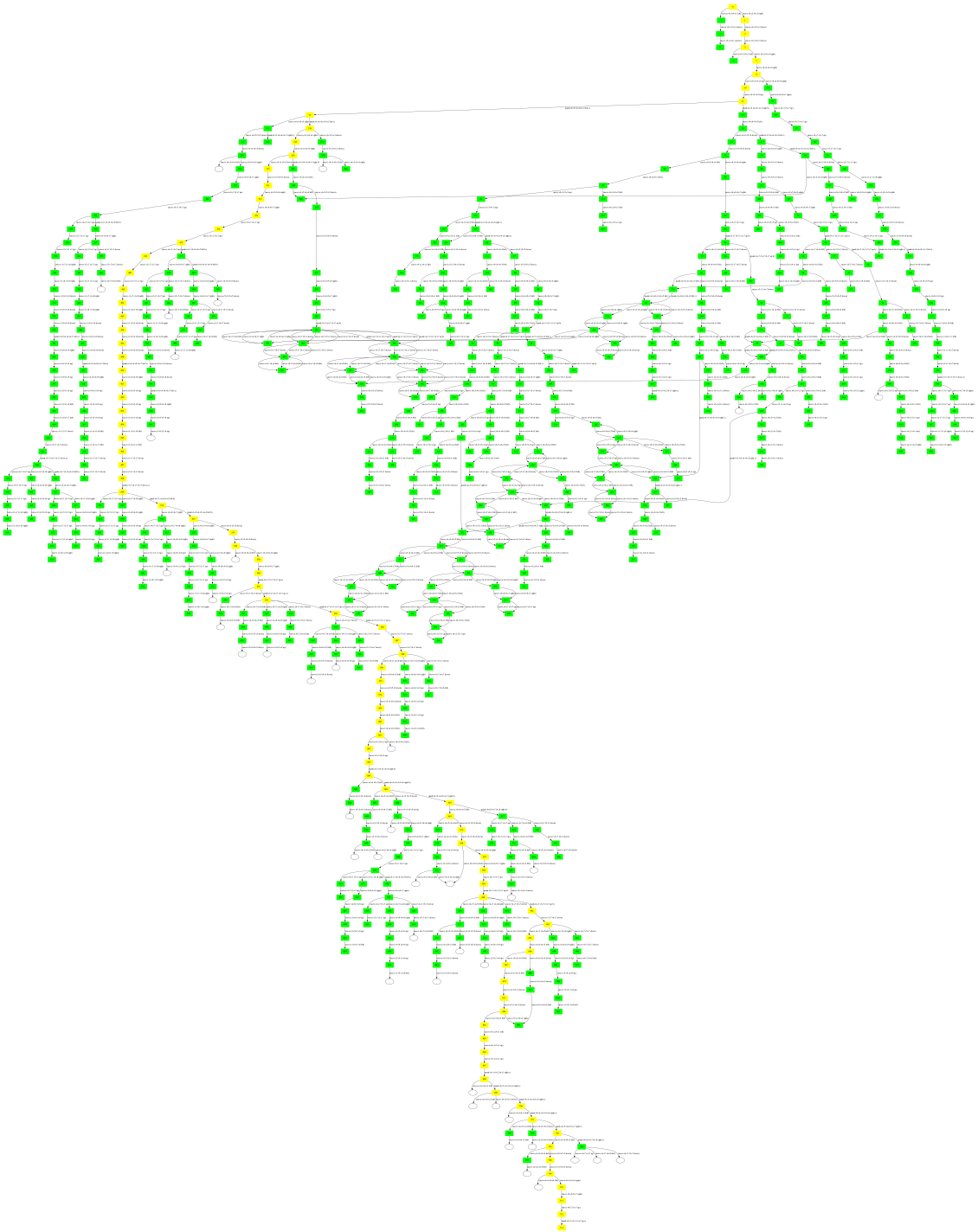
```
2014-10-05 00:20:18,758 INFO     Using plugin: visualizer
2014-10-05 00:20:18,758 INFO     problem: /home/tox/git/HW1_Team3/Code/Worlds/world3_proble.pddl
2014-10-05 00:20:18,758 INFO     using search: greedy_best_first_search
2014-10-05 00:20:18,758 INFO     using heuristic: hFFHeuristic
2014-10-05 00:20:18,758 INFO     using transition reduction: NoneTransitionPruning
2014-10-05 00:20:18,758 INFO     using state reduction: NoneStatePruning
2014-10-05 00:20:18,758 INFO     Parsing domain /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
domFile: /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
2014-10-05 00:20:18,760 INFO     Parsing problem /home/tox/git/HW1_Team3/Code/Worlds/world3_proble.pddl
2014-10-05 00:20:18,776 INFO     7 Predicates parsed
2014-10-05 00:20:18,776 INFO     2 Actions parsed
2014-10-05 00:20:18,776 INFO     95 Objects parsed
```

```
2014-10-05 00:20:18,776 INFO     0 Constants parsed
2014-10-05 00:20:18,776 INFO     Grounding start: sokoban_problem
2014-10-05 00:20:20,810 INFO     Relevance analysis removed 17 facts
2014-10-05 00:20:20,810 INFO     Grounding end: sokoban_problem
2014-10-05 00:20:20,810 INFO     135 Variables created
2014-10-05 00:20:20,810 INFO     192 Operators created
2014-10-05 00:20:20,810 INFO     Search start: sokoban_problem
2014-10-05 00:20:20,811 INFO     Initial h value: 19.000000
2014-10-05 00:20:22,436 INFO     Goal reached. Start extraction of solution.
2014-10-05 00:20:22,436 INFO     714 Nodes expanded
2014-10-05 00:20:22,481 INFO     Search end: sokoban_problem
2014-10-05 00:20:22,481 INFO     Wall-clock search time: 1.67087
2014-10-05 00:20:26,493 INFO     Plan length: 89
2014-10-05 00:20:26,494 INFO     validate could not be found on the PATH so the plan can not be validat
```

Looks like it only took 1.67 sec and 714 expanded nodes to find a path of the same length. That's cool! We'd have to compare the solution directly to see if the are in effect equivalent, but if we assume that all moves have the same cost value, such that walking costs the same as pushing, then we already know these two paths to goal have similar total costs.

In [29]:
```
Image("Worlds-gbf-hff-world3_proble.png")
```

Out[29]:



Ok, now let's try the challenge problem as show below:

```
In [8]: world    = "Worlds/world4.txt"
        problem  = "Worlds/world4_proble.pddl"
        solution = "Worlds/world4_proble.pddl.soln"
        # sokoban = Sokoban(world, labeled_boxes = False)
        # sokoban.writeProblem(problem)
```

```
In [23]: %%bash -s "cat" "$world"
         $1 $2
```

```
XXXXXXX
XXX__XX
XXXb_XX
X_b__rX
X_ggg_X
XXb__XX
XX__XXX
XXXXXXX
```

We'll use our script to generate most of the problem PDDL file first, but we'll have to do a bit of tweaking to get the solver to understand our intent. First off, we'll have to give each box a unique label so that the planner can keep track of the three different boxes:

```
; Init block objects and locations
(block a)
(block-at a r2-c3)
(block b)
(block-at b r3-c2)
(block c)
(block-at c r5-c2)
```

Then we'll need to modify the goal. Originally, or more efficiently the goal was expressed in this form:

```
; Define goal states
    (:goal
        (and
            (not (empty r4-c2)) (not (robot-at r4-c2))
            (not (empty r4-c3)) (not (robot-at r4-c3))
            (not (empty r4-c4)) (not (robot-at r4-c4))
        )
    )
```

However, as we found out,
>An exception is quantified goals in ADL domains, where of course the quantified variables may be used within the scope of the quantifier. However, even some planners that claim to support ADL do not allow quantifiers in goals.)" source

So, we can instead simple express the goal state alternativly by stating the ocupancy of every object excluding those of the blocks:

```
; Define goal states
    (:goal
        (and
            (empty r1-c3)
            (empty r1-c4)
            (empty r2-c3)
            (empty r2-c4)
            (empty r3-c1)
            (empty r3-c2)
            (robot-at r3-c3)
            (empty r3-c4)
            (empty r3-c5)
            (empty r4-c1)
            (empty r4-c5)
```

```
            (empty r5-c2)
            (empty r5-c3)
            (empty r5-c4)
            (empty r6-c2)
            (empty r6-c3)
        )
    )
```

Ok, this problem seams to have a terable branching factor, so lets skip breadth first seach here a try our HFF and GBF search instead.

```
In [20]: %%bash -s "$pyperplan" "$domain" "$problem"
         $1 $2 $3 $4 -H hff -s gbf
```

```
2014-10-06 10:45:07,536 INFO     problem: /home/tox/git/HW1_Team3/Code/Worlds/world4_proble.pddl
2014-10-06 10:45:07,536 INFO     using search: greedy_best_first_search
2014-10-06 10:45:07,536 INFO     using heuristic: hFFHeuristic
2014-10-06 10:45:07,536 INFO     using transition reduction: NoneTransitionPruning
2014-10-06 10:45:07,536 INFO     using state reduction: NoneStatePruning
2014-10-06 10:45:07,536 INFO     Parsing domain /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
domFile: /home/tox/git/HW1_Team3/Code/Worlds/sokoban_domain.pddl
2014-10-06 10:45:07,538 INFO     Parsing problem /home/tox/git/HW1_Team3/Code/Worlds/world4_proble.pddl
2014-10-06 10:45:07,549 INFO     7 Predicates parsed
2014-10-06 10:45:07,549 INFO     2 Actions parsed
2014-10-06 10:45:07,549 INFO     63 Objects parsed
2014-10-06 10:45:07,549 INFO     0 Constants parsed
2014-10-06 10:45:07,549 INFO     Grounding start: sokoban_problem
2014-10-06 10:45:08,266 INFO     Relevance analysis removed 27 facts
2014-10-06 10:45:08,266 INFO     Grounding end: sokoban_problem
2014-10-06 10:45:08,266 INFO     95 Variables created
2014-10-06 10:45:08,266 INFO     150 Operators created
2014-10-06 10:45:08,267 INFO     Search start: sokoban_problem
2014-10-06 10:45:08,267 INFO     Initial h value: 10.000000
2014-10-06 10:45:09,617 INFO     Goal reached. Start extraction of solution.
2014-10-06 10:45:09,618 INFO     1450 Nodes expanded
2014-10-06 10:45:09,618 INFO     Search end: sokoban_problem
2014-10-06 10:45:09,618 INFO     Wall-clock search time: 1.35181
2014-10-06 10:45:09,619 INFO     Plan length: 84
2014-10-06 10:45:09,620 INFO     validate could not be found on the PATH so the plan can not be validat
```

Impressive, in about 1.3 sec with 1450 nodes explored we've found a solution with a path length of 84. One thing to note here however is that the performance here varies a little, previous runs have been seen to range from 1450 to 7000 explored nodes, ranging from 1.3 to 6 sec runtimes. For completeness, let's show the solution here and take a look at some expanded trees.

```
In [24]: %%bash -s "cat" "$solution"
         $1 $2
```

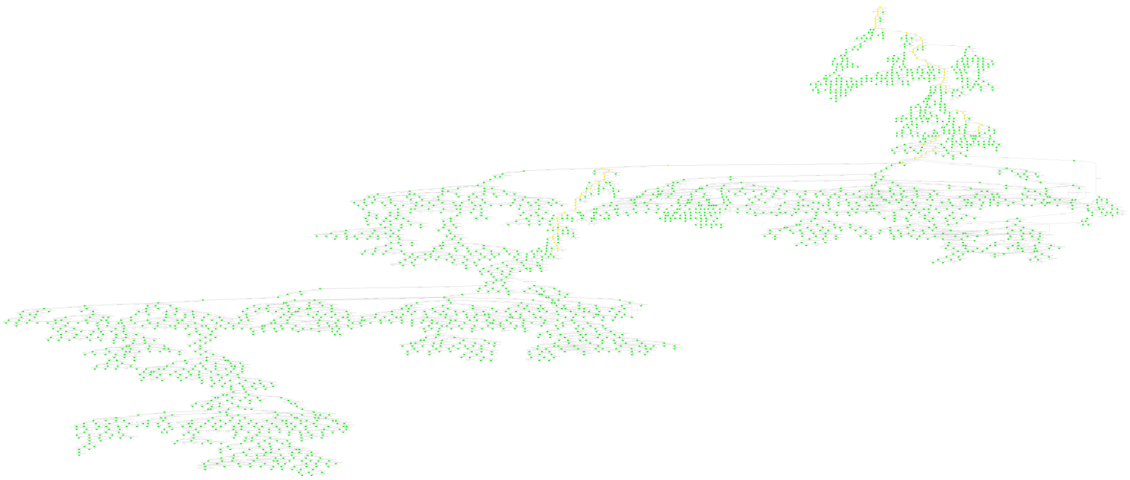
```
(move r3-c5 r3-c4 left)
(move r3-c4 r3-c3 left)
(move r3-c3 r4-c3 down)
(move r4-c3 r4-c2 left)
(move r4-c2 r4-c1 left)
(move r4-c1 r3-c1 up)
(push r3-c1 r3-c2 r3-c3 right b)
```

```
(push r3-c2 r3-c3 r3-c4 right b)
(move r3-c3 r4-c3 down)
(move r4-c3 r5-c3 down)
(move r5-c3 r6-c3 down)
(move r6-c3 r6-c2 left)
(push r6-c2 r5-c2 r4-c2 up c)
(move r5-c2 r5-c3 right)
(move r5-c3 r4-c3 up)
(move r4-c3 r3-c3 up)
(move r3-c3 r3-c2 left)
(move r3-c2 r3-c1 left)
(move r3-c1 r4-c1 down)
(push r4-c1 r4-c2 r4-c3 right c)
(move r4-c2 r5-c2 down)
(move r5-c2 r5-c3 right)
(move r5-c3 r5-c4 right)
(move r5-c4 r4-c4 up)
(move r4-c4 r4-c5 right)
(move r4-c5 r3-c5 up)
(push r3-c5 r3-c4 r3-c3 left b)
(push r3-c4 r3-c3 r3-c2 left b)
(move r3-c3 r3-c4 right)
(move r3-c4 r2-c4 up)
(move r2-c4 r1-c4 up)
(move r1-c4 r1-c3 left)
(push r1-c3 r2-c3 r3-c3 down a)
(move r2-c3 r2-c4 right)
(move r2-c4 r3-c4 down)
(move r3-c4 r4-c4 down)
(move r4-c4 r5-c4 down)
(move r5-c4 r5-c3 left)
(move r5-c3 r5-c2 left)
(move r5-c2 r4-c2 up)
(push r4-c2 r4-c3 r4-c4 right c)
(push r4-c3 r3-c3 r2-c3 up a)
(move r3-c3 r3-c4 right)
(move r3-c4 r2-c4 up)
(move r2-c4 r1-c4 up)
(move r1-c4 r1-c3 left)
(push r1-c3 r2-c3 r3-c3 down a)
(push r2-c3 r3-c3 r4-c3 down a)
(push r3-c3 r4-c3 r5-c3 down a)
(move r4-c3 r4-c2 left)
(move r4-c2 r4-c1 left)
(move r4-c1 r3-c1 up)
(push r3-c1 r3-c2 r3-c3 right b)
(move r3-c2 r4-c2 down)
(move r4-c2 r4-c3 right)
(push r4-c3 r3-c3 r2-c3 up b)
(move r3-c3 r3-c4 right)
(move r3-c4 r3-c5 right)
(move r3-c5 r4-c5 down)
(push r4-c5 r4-c4 r4-c3 left c)
(push r4-c4 r4-c3 r4-c2 left c)
```

```
(move r4-c3 r4-c4 right)
(move r4-c4 r5-c4 down)
(push r5-c4 r5-c3 r5-c2 left a)
(move r5-c3 r4-c3 up)
(move r4-c3 r3-c3 up)
(move r3-c3 r3-c2 left)
(move r3-c2 r3-c1 left)
(move r3-c1 r4-c1 down)
(push r4-c1 r4-c2 r4-c3 right c)
(push r4-c2 r4-c3 r4-c4 right c)
(move r4-c3 r5-c3 down)
(move r5-c3 r6-c3 down)
(move r6-c3 r6-c2 left)
(push r6-c2 r5-c2 r4-c2 up a)
(move r5-c2 r5-c3 right)
(move r5-c3 r4-c3 up)
(move r4-c3 r3-c3 up)
(move r3-c3 r3-c4 right)
(move r3-c4 r2-c4 up)
(move r2-c4 r1-c4 up)
(move r1-c4 r1-c3 left)
(push r1-c3 r2-c3 r3-c3 down b)
(push r2-c3 r3-c3 r4-c3 down b)
```

In [12]: `Image("Figures/Worlds-gbf-hff-world4_proble.png")`

Out[12]:

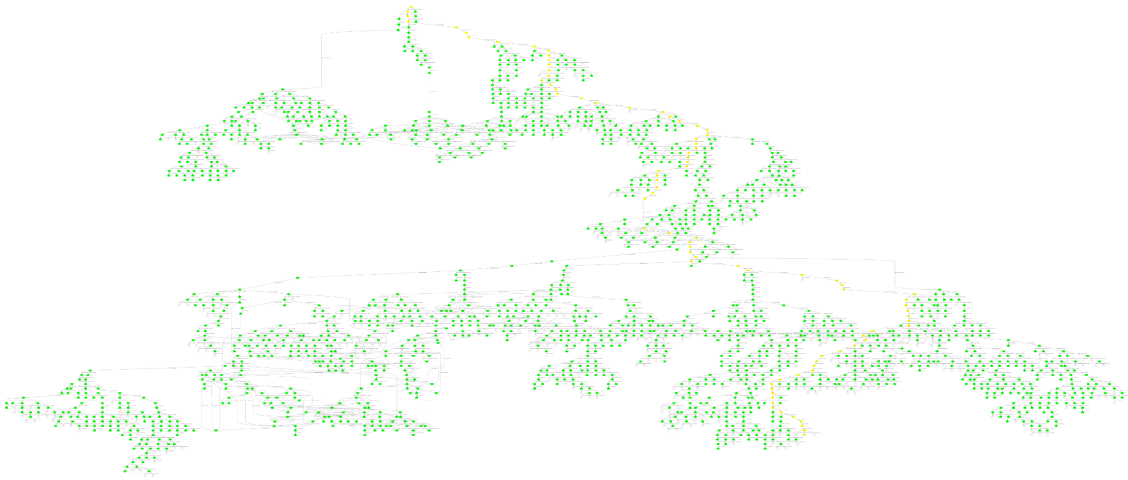


The figure above is from a particularly long run of about 6 sec with 7000 nodes explored. One interesting thing to note is that if you follow the yellow path to the goal state, you'll see it terminate in about the center of the frame of the figure. We can determine that the greedy search was first misguided into exploring farther into the state space before terminating and backtracking to the relevant branch, thus the reason for the lower left section of the tree extending past the final termination of the search at goal state.

Below was a more favorable search, taking about 1.5 sec and only expanding about 1500 nodes. We can see that the branching factor here is still quite formidable, and would have caused havoc for our earlier breadth first search approach.

In [16]: `Image("Figures/Worlds-gbf-hff-world4_proble-2.png")`

Out[16]:



1.2.2 b) Compare the performance of two planners on this domain. Which one works better? Does this make sense, why?

In general, and unsurprisingly, our GBF method combined with a HFF heuristic proves faster than the breadth first search. GBF does not guarantee optimality, unlike breadth first search, but then again what good is optimality if you do not have time or space resources to compute it?

Like many things, it really depends on the application. If this planning was for say a mobile game platform, where computation power is limited, and the user experience relies on a quick and responsive UI, then suboptimal GBF search might be just the tool to use. However, if the application was say for a logistics process, like for mailing and shipping with international freight, each additional move or action could be immensely expensive, lowering a company's bottom line. Such a large corporation may have use for a algorithm that guarantees optimality and completeness and would be willing to provide the computational power to assist in finding a better solution.

For these sokoban problems, I'd have to say that that GBF method method works better for a frustrated player who would like a quick answer or cheat solution manual from a computer for the current game and just level up to the the next sokoban challenge.

1.2.3 c) Clearly PDDL was not intended for this sort of application. Discuss the challenges in expressing geometric constraints in semantic planning.

Explicitly modeling the state of every element in the world is a tremendous overhead for semantic planning when trying to encode a geometric problem. Besides the initialization of world elements, i.e. there locations, types, and states, discretizing the state space to accommodate for a semantic planning can sometimes reduce dimension of the states but can also reduce the quality or dexterity of paths that would include continuous motions.

For instance, the fact that a human may not necessarily occupy the same volume or require the same clearance to move is lost in the current discretized world. Another thing is that the blocks are assumed to uniform in shape and scale, adding support for a multi-size category of blocks could be painful with a semantic planner. Such abstraction could be better supported and maintained by internalizing geometric constraints directly into the planner, rather than explicitly expressing every constraint directly.

1.2.4 d) In many cases, geometric and dynamic planning are insufficient to describe a domain. Give an example of a problem that is best suited for sematic (classical) planning. Explain why a semantic representation would be desirable.

For higher level task such as resource management for airlines, shipping companies, semantic planning still plays a big role in daily operation and upkeep. Much of the revival of AI form The Dark Ages of AI comes from the relaization from industy and govermental sponsors that basic computerized planning can afford maxumem effecientcy with minimal effort. The US Army now continues to improve it's internal logistics using widespread departmental planning.

> The Logistics Modernization Program (LMP) is one of the programs that stands at the center of the Army's business transformation initiatives. The LMP is a cornerstone of the Single Army Logistics Enterprise—an enterprise business solution that will enable vertical and horizontal integration at all levels of logistics across the Army. By modernizing both the systems and the processes associated with managing the Army's supply chain at the national and installation levels, the LMP will permit the planning, forecasting, and rapid order fulfillment that lead to streamlined supply lines, improved distribution, a reduced theater footprint, and a warfighter who is equipped and ready to respond to present and future threats. Source

Other agencies and corporations that rely heavily on logistic optimization like UPS or Amazon have many applications suited for semantic planning as the underlying properties.

1.3 3) Sokoban Challenge

You may now think that the difficulty of this problem has to do with PDDL. Your instructor seriously doubts that. Lets try an experiment. Define the Sokoban in any way that you like and write a computer program that solves it. Your instructor recommends you start with some kind of simple state space search and then see if you can make it faster. Test your planner on the sample problems in Figure 2 and any other examples you find interesting. Make sure that your planner addresses the following points: * The planner must be complete. * Your state representation should be efficient. * The planner should make an effort to be fast.

To give you feedback on how well you accomplished these objectives we conduct an informal competition for who can produce the most optimal planner for the 4th problem(Figure 2.(d)). Purportedly this is a difficult problem for humans to solve - have a go at it before you let your planner do the work! If your planner can solve it, include the solution and report the following pieces of information: * Computation time for Sokoban Challenge (and the other problems) * Number of steps in your plan * Number of states explored * Language used * Machine vitals

1.3.1 a) Give successful plans from your planner on the Sokoban problems in Figure 2 and any others.

For the sokoban challenge, we designed a planner that functions similar to the A* algorithm.

Language used: C++ Machine vitals: Intel® Core™ i7-4770 CPU @ 3.40GHz × 8 16 Gb RAM

```
Intel® Core™ i5 CPU @ 2.40GHz × 4
8 Gb RAM

Multithreading not used to compute time
All times computed on the i7 processor
```

Operating system compatibility: Ubuntu 14.04, Mac OS X 10.9.4
Compilers tested: Clang GCC
Required Libraries: Boost 1.54 +
For plans refer to Code/question_3/plans/plan_*.txt

```
In []: %%bash -s
       cd question_3/
```

```
mkdir build && cd build
cmake ..
make
# For question 1, 2, 3, and 4
./planner_state_space ../problems/q1.txt
./planner_state_space ../problems/q2.txt
./planner_state_space ../problems/q3.txt
./planner_state_space ../problems/q4.txt
```

1.3.2 b) Compare the performance of your planner to the PDDL planners you used in the previous problem. Which was faster? Why?

For 2.1,

Planner	Length	Time	Nodes Expanded
Our Planner	18	0.054	107
Blackbox	14	0.322	320
PyPlanner	14	0.040	95

Our planner performs better than blackbox in terms of nodes expanded and the time but the length of the plan is higher. The pyplanner does better and the reasons for this could be the heuristic function that is being used and some code optimization. It also looks like the pyplanner is computing the whole graph first and just timing the search whereas we measure execution time.

For 2.2,

Planner	Length	Time	Nodes Expanded
Our Planner	32	0.821	808
PyPlanner bfs	32	0.453	953
PyPlanner gbf	36	0.090	95

Our planner performs better than the greedy best first search of Pyplanner in terms of length but is undone by nodes expanded and the time taken. The breadth first search is computing a plan of the same length but expands more nodes. However it is faster. This could be because of the choice of heuristic and the way their graph is being computed.

For 2.3,

Planner	Length	Time	Nodes Expanded
Our Planner	105	82.606	6588
PyPlanner bfs	89	4.89	7867
PyPlanner gbf	89	1.79	714

Our planner finds a plan of higher length and expands less nodes but takes a lot more time. The pyplanner works a lot better for this particular problem. The corner search that we perform in this case does not help. Also, the heuristic chosen does not work too well for this problem.

Planner	Length	Time	Nodes Expanded
Our Planner	76	1.35	5302
PyPlanner bfs	NA	NA	NA
PyPlanner gbf	85	1.35	1450

Our planner finds a plan of lower length, the breadth first search never finds a plan and the greedy best first search never finds the solution. We expanded more nodes and take more time but our planner guarantees completeness.

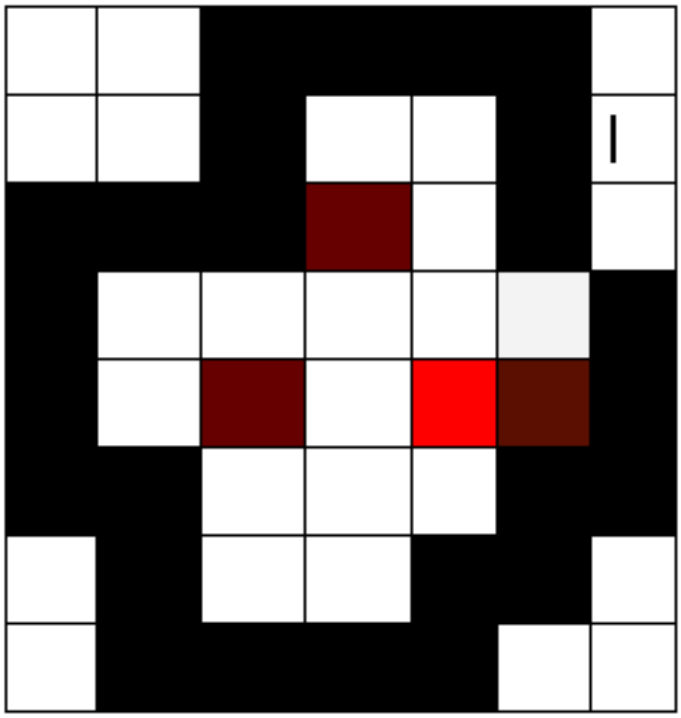
1.3.3 c) Prove that your planner was complete. Your instructor has a math background: a proof is a convincing argument. Make sure you address each aspect of completeness and why your planner satisfies it. Pictures are always welcome.

The planner is A* based so guarantees completeness. Every possible state is added to a frontier list. The speed up trick tried was to prune the tree at every point where the state is valid but the solution is not reachable from that state.

$$Heuristic(cost - to - go) = \sum (box - goal) + \sum (robot - box)$$

```
In [51]: Image("Figures/lost_cause.png")
```

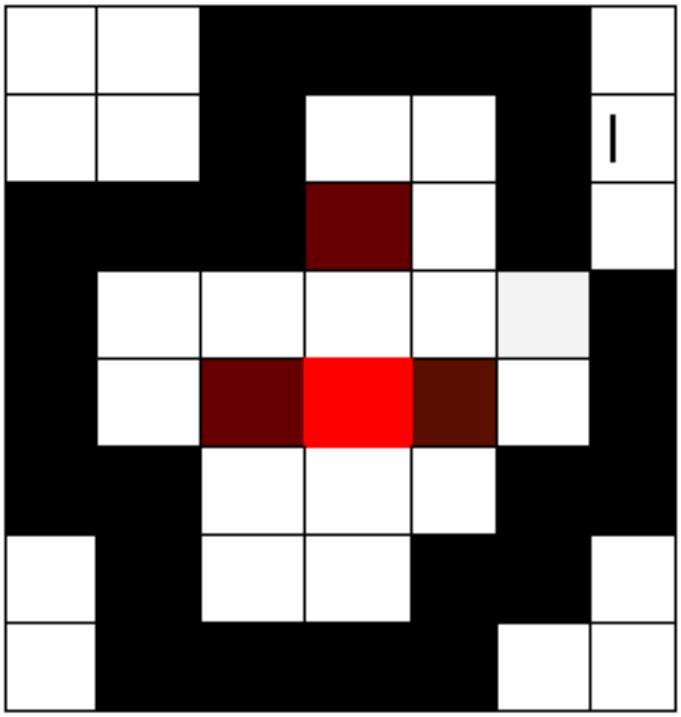
Out[51]:



Lost Cause figure

```
In [53]: Image("Figures/start_state.png")
```

Out[53]:



Start State figure

In the figures, let the red boxes be the robot, and the brown be the boxes, black occupied spaces and the white empty spaces.

For instance the start state figure shows a state from where the state in the lost cause figure is reachable but the state on the right need not contribute to the tree since the solution is not reachable from there.

This means that the completeness is still guaranteed since even though every possible state wont be explored, the solution cannot exist along the branch which is being pruned. This means that completeness is still guaranteed.

Even if the cost is wrong, since every possible move goes into the frontier if a solution exists the planner will find it.

1.3.4 d) What methods did you use to speed up the planning? Give a short description of each method and explain why it did or didnt help on each relevant problem.

To speed up the planning we used a forward search on the boxes near corners to prune the tree of possible states. This helped a lot with the 4th problem but found the 3rd problem problematic.

The lists were stored as fibonacci heaps so that the heaps were sorted and kept the data structures linear in most operations. This performs better than an unsorted list as the complexity grows with every iteration where the sort is required.

STL find was used to streamline the iteration of the list variables to check for uniqueness. The assignment for the structures was made more optimal by overloading operators. Guaranteeing uniqueness in the lists 'frontier' and 'explored' keeps the state from growing very large. The STL find function performs about 5 times faster than iterating over the heap in order.

The heurisitic chosen was modified slightly to aggressively move towards the boxes at the start by varying the gains and this caused some speed up in the solutions.

The speedups only worked only up to a scale but as the problem got more and more complex the solver needed a lot more space to perform the computations necessary.

The pyplanner still works faster but both but the best first search which would guarantee completeness does not scale up to more complex problems whereas our planner does.

1.4 4) Towers of Hanoi Revisited

Having struggled to solve Imabayashi-sans puzzles, maybe we have a better feel for what planning is all about. Lets take another trip out to Hanoi. With modern automation, the monks aspire to even greater challenges. They can now solve Hanoi tower problems where there are 10 discs on the first tower. Can your planners from Problem 1 do this? Change the PDDL specification so that you can solve the problem with 10, and even 12 discs.

```
In [27]: Image("Figures/Towers of Hanoi.png")
```

Out[27]:

1.4.1 a) Give successful plans from at least one planner with 10 and 12 discs.

The PDDL codes for the two scenarios are in this repo under hanoi-domain.pddl (as provided), hanoi10.pddl, and hanoi12.pddl. First, it was confirmed that Blackbox and VHPOP could not solve these problems with reasonable time/memory. Both planners had to be killed after using up the computer's entire available RAM after about half an hour. A record of the Blackbox attempt at a solution can be seen in hanoi10_blackbox, where it is explicit that the sub-goals are mutually exclusive and graphplan must be called. However, even this is not sufficient, as the planner needs to search every graph of a given depth before ruling it out and augmenting the graph by an additional level, making it take far too long to reach the 1023 steps minimally necessary to solve the ten disk problem.

```
In [55]: %%bash -s "$blackbox" "Towers/hanoi-domain.pddl" "Towers/hanoi10.pddl"
         $1 -o $2 -f $3
```

```
Process is terminated.
```

```
In []: %%bash -s "$vhpop" "Towers/hanoi-domain.pddl" "Towers/hanoi10.pddl"
       $1  -f LCFR -l 10000 -f MW -l unlimited $2 $3
```

Therefore, the FF planner was chosen instead (https://fai.cs.uni-saarland.de/hoffmann/ff/) as it first identifies graph depth in a relaxed search and then backs out the solution. FF produced successful solutions very quickly (less than 1 s for 10 disks and less than 10 s for 12 disks); these plans are given in hanoi10_sol and hanoi12_sol.

```
In []: %%bash -s
       ./question_5/FF/FF-v2.3/ff -o Towers/hanoi-domain.pddl -f Towers/hanoi10.pddl
```

```
In []: %%bash -s
       ./question_5/FF/FF-v2.3/ff -o Towers/hanoi-domain.pddl -f Towers/hanoi12.pddl
```

In [25]: Image("Figures/Fig 4.png")

Out[25]:

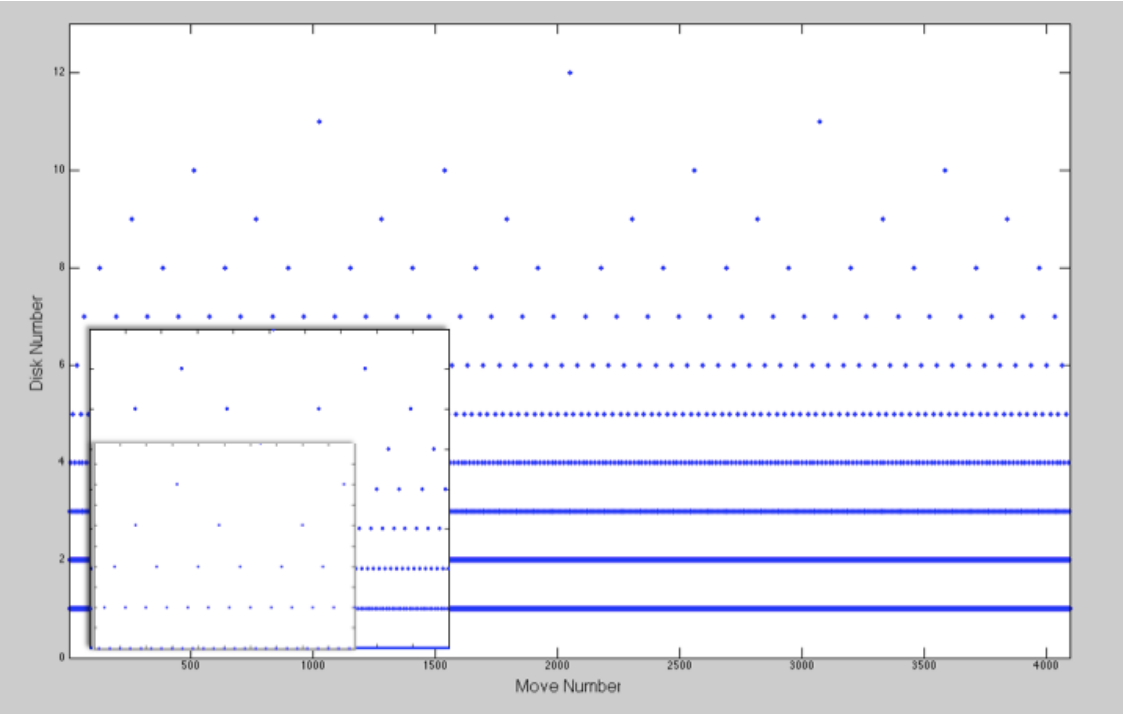


Figure 4.1: Successful plan for solving the twelve-disk Tower of Hanoi problem. When the moved disk is plotted against move number the recursive structure of the overall problem is clearly illustrated by looking at the sub-structure of the overall solution.

1.4.2 b) Do you notice anything about the structure of the plans? Can you use this to increase the efficiency of planning for Towers of Hanoi? Explain.

The solutions have a very clear recursive structure that can be seen most clearly when plotted (see Fig. 4.1). The structure is both recursive and symmetric, every move that is made in deconstructing the first tower is required to build the second tower. Tower size does not matter beyond whether there is an even or odd number of disks. Once that is known the "algorithm" is very simple. For instance, with an even number of disks the algorithm is to move the smallest disk and then the second smallest disk, iteratively. If possible, you first move the appropriate disk from the left tower to the center, then from the left tower to the right tower, and then from the center to the right. This same algorithm is repeated, skipping non-possible/illegal steps (putting a larger disk on a smaller one) as necessary. Thus, the efficiency could be greatly increased to a simple three-step algorithm that just checks for legal moves as well as the parity of the disks.

1.4.3 c) In a paragraph or two, explain a general planning strategy that would take advantage of problem structure. Make sure your strategy applies to problems other than Towers of Hanoi. Would such a planner still be complete?

The algorithm given above would make solving the Tower of Hanoi extremely easy, as no planning would be necessary. That algorithm solves any even-numbered Tower of Hanoi problem, and a similar algorithm exists for solving the puzzle with an odd number of disks. Therefore, the best way to generalize this problem is to use a planner, even a less powerful one like Blackbox, to solve the smaller problems (for 3, 4, 5 disks, etc) and then just remember/save that smaller solution so that it doesn't have to keep re-solving it. This would

reduce the number of nodes that the planner needs to search because it would only need to find a solution once for each additional disk.

This sort of planner will only work for recursive problems, so it would not solve every large, complex puzzles. However, the completeness of such a recursive "planner" is dependent on which "internal planner" (STRIPS, VHPOP, FF) is chosen, as the larger recursive "planner" just makes the "internal planner" faster (more optimal/more efficient) and does not guarantee any more completeness. The overall planner's completeness is equivalent to that of the "internal planner".

1.5 5) Towers of Hanoi with HTN planning

We can encode our domain knowledge in the HTN planning framework. Let's see if we can get some performance improvement by using the domain knowledge actively.

1.5.1 a) Formulate a HTN planning problem for the Towers of Hanoi.

HTN planning for tower of Hanoi can be formulated as a recursive algorithm. At each level of this algorithm, it decomposes moving a tower }of size k from source to destination into the problem of moving a tower of size $k-1$ and the moving the k^{th} disk. It can be easily visualized if we consider the tower of size $k-1$ as one big disk and the other k^{th} disk and the task is to move the k^{th} disk from source to destination. After this moving the tower of size $k-1$ can be again sub divided into the problem of moving the tower of size $k-2$ and moving the $(k-1)^{th}$ disk and so on. The recursion ends when the size of $k=1$. Algorithm 1 summarizes the algorithm.

In [41]:
```
Image('Figures/Algorithm 1.png')
```

Out[41]:

Algorithm 1 Move-Tower($disk, source, dest, spare$)

1: **if** disk == 0 **then**
2: Move disk from *source* to *dest*
3: **else**
4: $Move-Tower(disk-1, source, spare, dest)$
5: Move disk from *source* to *dest*
6: $Move-Tower(disk-1, spare, dest, source)$
7: **end if**

1.5.2 b) Describe the domain knowledge you encoded and resulting planning domain (i.e. primitive tasks, compound tasks, and methods) in detail.

The domain knowledge that we encoded is that the towers of hanoi problem can be solved by recursively splitting the main problem into two high level actions and a primitive action. The high level action moves a tower of smaller size followed by a primitive action of moving the bottom most disk. The high level actions are further refined until it reaches a primitive action. The primitive task is defined in Table 1 and the compound task is defined in Tables 2 and 3. Figure 1 shows the corresponding flowchart.

In [42]:
```
Image("Figures/tables.png")
```

Out[42]:

move(disk, source, destination)
precond: location(disk) == source
effect: location(disk) = destination

Table 1: Primitive Move disk Task

move_tower(disk, source, destination, spare)
• task: move_stack(disk, source, destination, spare)
• precond: disk > 0
• subtasks: <move_tower(disk-1, source, spare, destination),move(disk, source, destination), move_tower(disk - 1, spare, destination , source)>

Table 2: Compound move_stack task

move_tower(disk, source, destination, spare)
• task: move_disk(disk, source, destination, spare)
• precond: disk == 0
• subtasks: move(disk, source, destination)

Table 3: Compound move_disk task


```
In [29]: Image('Figures/RIP-Assignment1.png')
```

Out[29]:

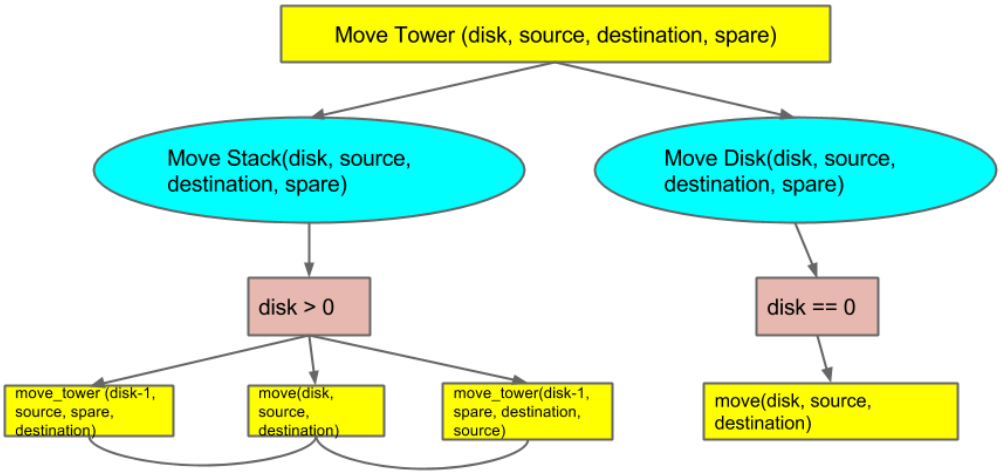


Figure 1: Algorithm flowchart for HTN

1.5.3 c) Solve your HTN problem for the cases with 3 to 12 discs (use SHOP/SHOP2 or whatever HTN planner available to you).

The code is attached. It can be executed using "'python towers_of_hanoi.py

```
In [46]: %%bash -s
         # you can just look at Towers/towers_of_hanoi_HTN_solution.txt to see the super long output
         # but the command below is how you would run the solver
         #python question_5/HTN/towers_of_hanoi.py 10
```

1.5.4 d) Solve the same cases with a non-HTN planner of your choice, and compare the results with (c).

The code and pddl files are is attached. It can be executed using

```
./ff -o ../hanoi-domain.pddl -f ../hanoi<num_disks>.pddl
```

1.5.5 e) Describe your observations and discuss about them.

As we can see from below figures, time taken and the number of states explored by the HTN planner is much less than FF planner. The number of steps taken in the final plan is the same for both HTN planner and FF planner.

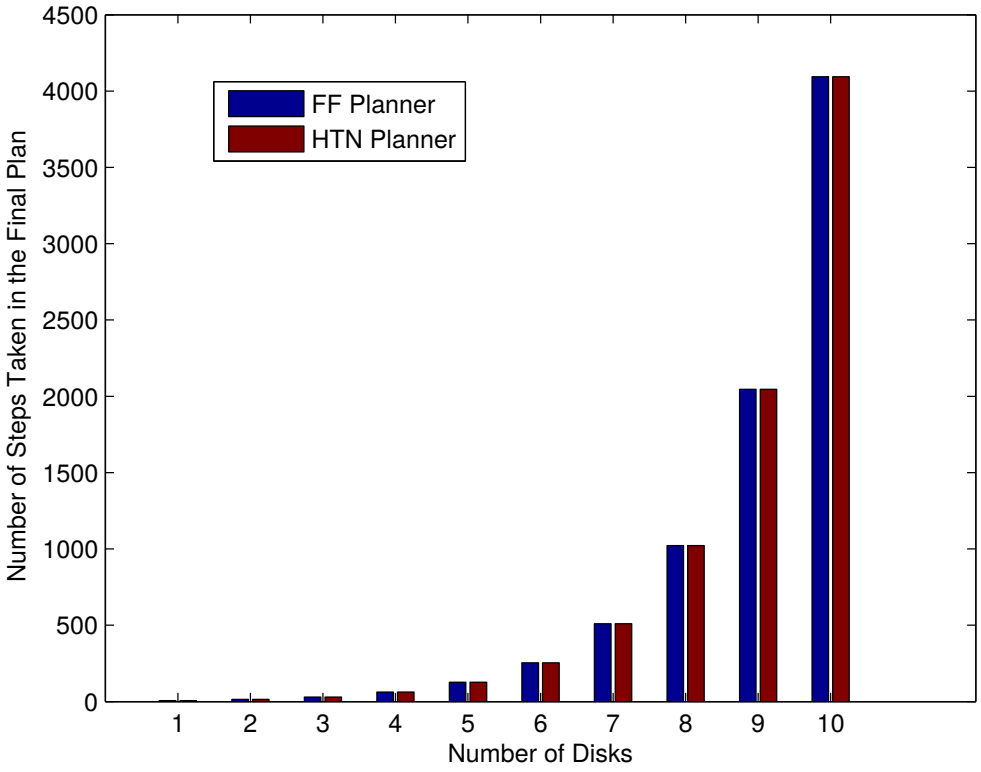
The reduction in the number of states and the time taken to explore the states is because of the domain knowledge that we encoded in HTN planning framework. This is because during each level of HTN planning it sub divides the problem into a sequence of easier primitive/compound tasks which it can run without failure. As a result the number of states explored which didn't fail are much less for HTN planning as compared to FF planning (or any other non HTN planning). Domain knowledge guides the planner towards the right search direction.

If we reduce the amount of domain knowledge, the planner has to explore more states to find a suitable plan which results in additional time taken to find the plan. However increasing the amount of domain

knowledge reduces generalizability of the planner to varied situations. So, there is a trade-off here in the amount of time taken to plan and the situations that planner can handle.

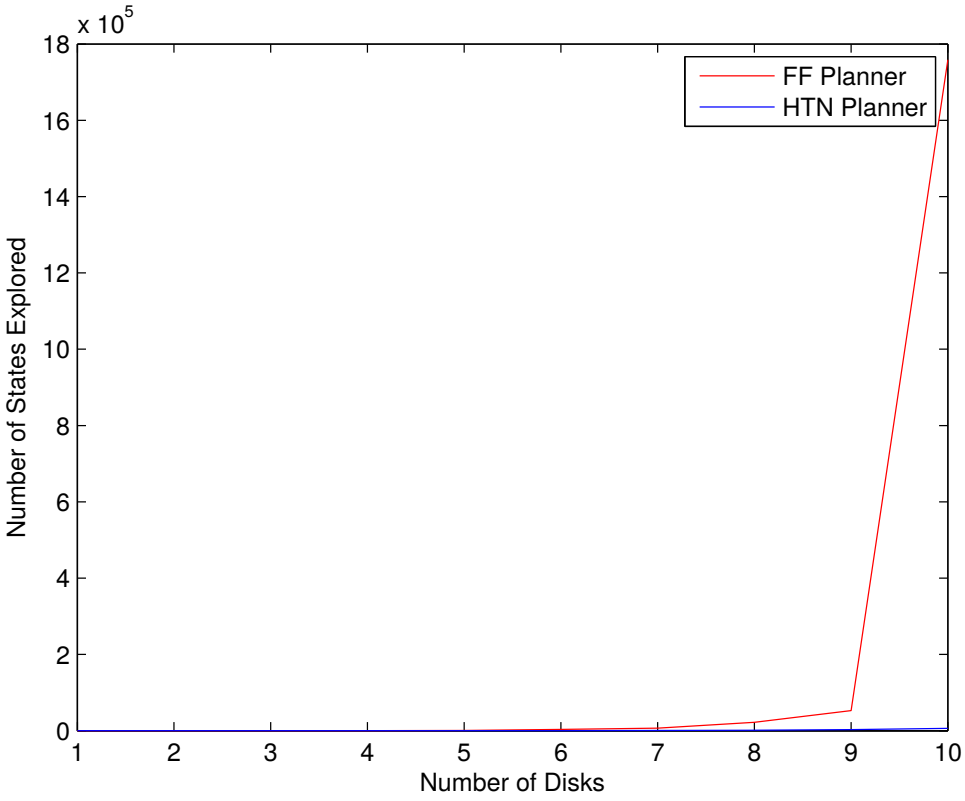
```
In [50]: SVG(filename="Figures/solution_size.svg")
```

```
Out[50]:
```



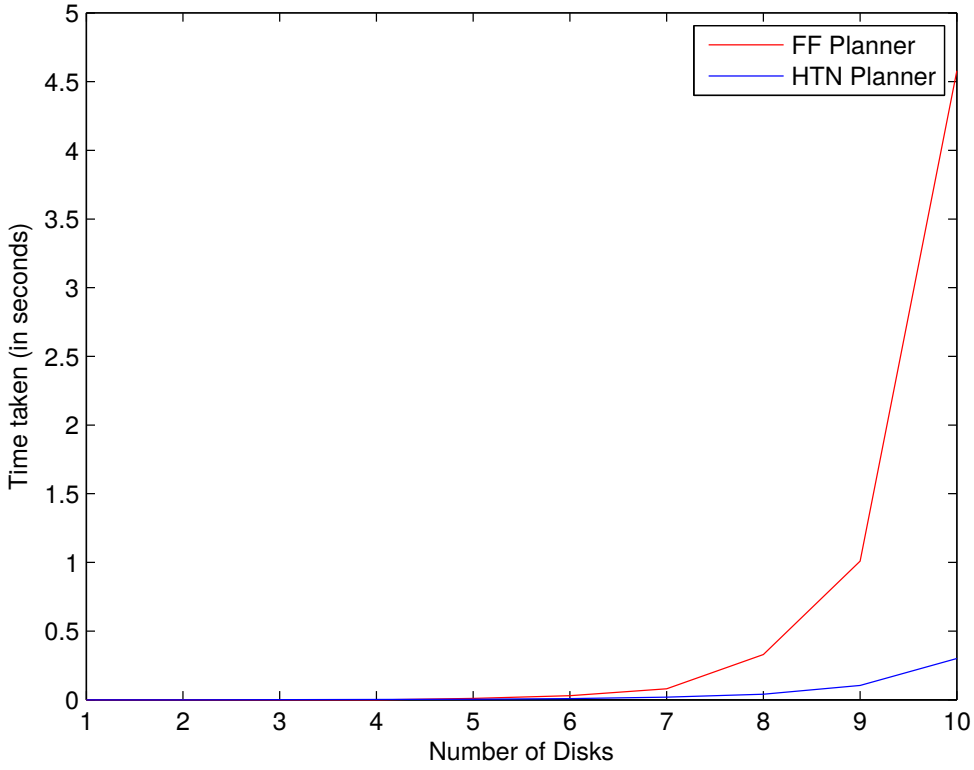

```
In [48]: SVG(filename="Figures/states_explored.svg")
```

```
Out[48]:
```




```
In [49]: SVG(filename="Figures/time_taken.svg")
```

Out[49]:


FF Planner
HTN Planner
Time taken (in seconds)
Number of Disks
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
1
2
3
4
5
6
7
8
9
10