Abhinav Rai
UIN – 320001645

CSCE 625 - AI
Homework #3
Due: Fri, Sep 30, 2011

Report

Problem Overview:

The problem here has a large branching factor. For each node, we can have '2 * num_blocks' possible configurations to move to. With such a large branching factor, DFS or BFS do not work fine. To add to the difficulty, the problem is inherently multi-dimensional, i.e. We want to move the final configuration to a state where all the nodes have adjacent values in sorted order, and we want all the nodes to be moved to the final stack. So it is a choice of 'block ordering in a stack' + 'relative configurations of the stacks'.

So we need to apply favourable heuristics to our search to converge the iterations to the goal node quickly.

Heuristics applied and observations:

Key_Ideas:

g_cost : cost accrued so far in the iterations.
h_cost : heuristic cost of a node to reach the goal node
cost = g_cost + h_cost
h_cost(goal) = 0

- h0 : First run employed search with 'heuristic cost = 0', which simulates the uniform cost search (breadth first) from any random state to goal state. With the large branching factor we had (2n, where n is the number of blocks), h0 search takes a large number of runs, and runs out of maximum tries to find out the goal for values of n > 5.

- h1 : The idea employed here was, that if any stack contains two nodes with adjacent values, that pair can be moved with less cost to achieve the goal state. So **for any two nodes with adjacent values, we subtract the h_cost by 1**. This turned out to be a lot better. But sometimes the configuration advantage is nullified by the g_cost, i.e. If a better node has a higher g_cost, then in total cost priority, we fail to choose it as the frontier node.

- H2: From h1, we saw that the adjacent value node heuristics helps find a node faster. So I gave more weightage to that configuration, i.e. If we find such a configuration, **for each adjacent value pair, we decrease h_cost by 4.**

  This way, we can take advantage of the low h_cost and it does not get significantly affected from higher g_costs. This way, I was able to get to the goal node in fairly less iterations.

  Another intuition was, that in the goal node, all following nodes have a higher value than the previous one. So if a node has a higher value node following a lower value node, we decrease h_cost by 2.

To make the h_cost of goal 0, we start h_cost with a number 4 * 'num_blocks – 1'. The goal node contains 'num_blocks – 1' adjacent value pairs, and for each such pair, we decrease the h_cost by 4, eventually resulting in a cost 0 for goal node.

Heuristics/num_blocks	h0	h1	h2
3	35	26	3
4	110	62	5
5	2460	288	6
6	Test limit exceeded	542	8
7	“”	Above 2000	11
8	“”	Test limit exceeded	13
9	“”	“”	13
10	“”	“”	15
12	“”	“”	17
15	“”	“”	21

(avg. num_goal_tests)

Sample Outputs:

h2: n = 8

```
abhinav@ubuntu:~/Downloads$ python blocks.py 8
randomly generated stacks of blocks:
4
7
5
2   6
3   8   1

('cost = ', 24, 'g = ', 0)
4
7
5
2   6
3   8   1

('cost = ', 19, 'g = ', 1)
    4
    7
    5
2   6
3   8   1

('cost = ', 16, 'g = ', 2)
```

```
    2
    3
    4
    7
    5
    6
    8   1

('cost = ', 13, 'g = ', 3)
    1
    2
    3
    4
    7
    5
    6
    8

('cost = ', 12, 'g = ', 4)
        1
        2
    5   3
    6   4
    8   7


('cost = ', 11, 'g = ', 5)
    1
    2
    3
    4
    5
    6
    8   7

('cost = ', 10, 'g = ', 6)
        1
        2
        3
        4
        5
        6
    8   7

('cost = ', 7, 'g = ', 7)
    1
    2
    3
    4
    5
    6
    7
    8

('cost = ', 8, 'g = ', 8)
        1
        2
        3
        4
```

```
        5
        6
        7
        8

goal found.
('g =', 8, 'h =', 0)
('num_goal_test =', 8)
```

h2: n = 9:

```
abhinav@ubuntu:~/Downloads$ python blocks.py 9
randomly generated stacks of blocks:
7
3
8
4   9   1
6   5   2

('cost = ', 30, 'g = ', 0)
7
3
8
4   9   1
6   5   2

('cost = ', 25, 'g = ', 1)
    7
    3
    8
4   9   1
6   5   2

('cost = ', 22, 'g = ', 2)
    4
    6
    7
    3
    8
    9   1
    5   2

('cost = ', 21, 'g = ', 3)
    1
    2
    4
    6
    7
    3
    8
    9
    5

('cost = ', 20, 'g = ', 4)
        1
        2
        4
        6
        7
        3
```

```
        8
    5   9

('cost = ', 19, 'g = ', 5)
        6
    1   7
    2   3
    4   8
    5   9

('cost = ', 16, 'g = ', 6)
        1
        2
        4
        5
        6
        7
        3
        8
        9

('cost = ', 15, 'g = ', 7)
    1
    2
    4
    5   3
    6   8
    7   9

('cost = ', 14, 'g = ', 8)
        1
    4   2
    5   3
    6   8
    7   9

('cost = ', 13, 'g = ', 9)
    1
    2
    3
    4
    5
    6   8
    7   9

('cost = ', 10, 'g = ', 10)
        1
        2
        3
        4
        5
        6
        7
        8
        9

goal found.
('g =', 10, 'h =', 0)
('num_goal_test =', 10)
```

Appendix:

Source Code:

```
import random
import sys
from copy import deepcopy
from heapq import heappush, heappop

def find(f, seq):
  """Return first item to be found in the list matching with 'item'."""
  for item in seq:
    if f == item:
      return item
  return None

def print_blocks(stack_list):
    """prints stacks of blocks vertically"""
    max_blocks = 0
    for i in range(len(stack_list)):
        max_blocks = max(max_blocks, len(stack_list[i]))

    array = []
    for i in range(max_blocks):
        temp_list = []
        for j in range(len(stack_list)):
            k = max_blocks - i - 1
            if k < 0 or k >=len(stack_list[j]):
                temp_list.append(' ')
            else:
                temp_list.append(stack_list[j][k])

        array.append(temp_list)

    for i in range(max_blocks):
        print('   '.join(map(str, array[i])))
    print('\n')

def gen_children(stack_list):
    """ generates children thru valid moves from a stack list configuration"""
    list_of_children = []
    sl = stack_list
    for i in range(len(sl)):
        cur_stack = sl[i]
        len_cur = len(cur_stack)
        if not len_cur:
            continue

        for j in range(len_cur):
            for k in range(len(sl)):
                if k != i:
                    new_child = deepcopy(sl)
                    count = 0
                    for l in cur_stack[len_cur-j-1:]:
                        new_child[k].append(l)
                        count +=1
                    for p in range(count):
                        new_child[i].pop()
```

```
                list_of_children.append(new_child)

    return list_of_children

def gen_random(num_blocks):
    a = range(1,num_blocks+1)
    random.shuffle(a)
    f = random.randrange(0,num_blocks)
    s = random.randrange(f,num_blocks)
    return[a[:f],a[f:s],a[s:]]

class BlocksWorld(object):
    def __init__(self, num_blocks=8):
        self.num_blocks = num_blocks
        self.goal = [[],[],range(1,num_blocks+1)]
        self.goal[2].reverse()
        self.goal_tests = 0

        self.stack_list = gen_random(num_blocks)

    def h0(self, sl):
        return 0

    def h1(self, sl):
        h_cost = (self.num_blocks - 1)
        for i in range(len(sl)):
            cur_stack = sl[i]
            temp = 0
            for j in cur_stack:
                if temp:
                    if j == temp-1:
                        h_cost -= 1
                temp = j
        return h_cost

    def h2(self, sl):
        """ given a stack list configuration, returns the heuristic cost.
        h_cost(goal) = 0
        Closeness and thus lower cost of a config is based on the assumption,
        that if the next node in a stack is 'previous node + 1', such nodes are
        nearer to the goal. """
        # for each successive node in order, we'll reduce the cost by 4,
        #initialize cost to a higher value, so that for goal node,
        #ultimate h_cost comes out to be zero.
        h_cost = (self.num_blocks - 1) * 4
        for i in range(len(sl)):
            cur_stack = sl[i]
            temp = 0
            for j in cur_stack:
                if temp:
                    if j == temp-1:
                        h_cost -= 4
                    else:
                        if j < temp:
                            h_cost -= 2
                        else:
                            h_cost += 2
                temp = j
        return h_cost
```

```
    def a_star(self, calc_heuristics = h2):
        h = []
        seen = []
        g_cost = 0
        new_node = self.stack_list
        h_cost = calc_heuristics(new_node)
        cost = g_cost + h_cost
        heappush(h, (cost, g_cost, new_node))

        while len(h):
            (cost, g_cost, new_node) = heappop(h)
            seen.append(new_node)
            print('cost = ', cost, 'g = ', g_cost)
            print_blocks(new_node)

            if new_node == self.goal:
                print('goal found.')
                print('g =',g_cost,'h =', calc_heuristics(self.goal))
                print('num_goal_test =',self.goal_tests)
                break
            self.goal_tests +=1

            if self.goal_tests > 5000:
                print('max num goal tests exceeded.')
                return

            child_list = gen_children(new_node)
            g_cost+= 1
            for node in child_list:
                if new_node == self.goal:
                    print('goal found.')
                    print('g =',g_cost,'h =', calc_heuristics(self.goal))
                    print('num_goal_test =',self.goal_tests)
                    print_blocks(new_node)
                    break

                if not find(node, seen):
                    h_cost = calc_heuristics(node)
                    cost = g_cost + h_cost
                    heappush(h, (cost, g_cost, node))

def main():
    if len(sys.argv) != 2:
        print('usage: %s <num_blocks>' % sys.argv[0])

    if len(sys.argv) > 1:
        a = BlocksWorld(int(sys.argv[1]))
    else:
        a = BlocksWorld()

    print('randomly generated stacks of blocks:')
    print_blocks(a.stack_list)
    a.a_star(a.h2)

    return

if __name__ == '__main__':
    main()
```