TDT4136 Logic and Reasoning Systems

Jørgen Grimnes

Assignment 3

Fall 2013

1 A* algorithm

Implementation I've implemented the A* algorithm from scratch in *Python* to make it easier to read and understand the implementation. This implementation makes us of *PriorityQueues* to quickly decide which node has the currently best estimate for begin (close to) a goal state.

Generalized I've made a generalized implementation of the A* algorithm. The *class AstarNode* contains the structural important variables and serves as an abstract A*-node. Then by implementing the *class Node*, you will be able to quickly adapt this A* implementation. There are also two different methods that has to be implemented: *distance* and *heuristic*. Distance() is how the script should calculate the degree of difference between a parent and a child node. The heuristic() method should return an optimistic estimate as to how far away from the goal state the current node is.
The method *generate children* also has to be updated.

Speed-ups This Python script has focused on speed, rather than memory usage. There are no open or closed lists, they are implied through the boolean variables. This gets rid of the very expensive method of checking for list membership (compared to checking a single variable). To force this A* implementation into becoming a Depth-first search, we would only have to give each successing child node a lower f-score than the current nodes. This is achievable by using a semistatic decreasing f-score, and subtract a small value from this score before returning it to the given node.

$$Fscore_{initial} = 10000$$
$$Fscore_{nextnode} = Fscore_{initial} - \Delta_{small}$$
$$Fscore_{node} = Fscore_{nextnode}$$

To act as a breadth-first search, we would add a small delta to the f-score instead.

$$Fscore_{initial} = 0$$
$$Fscore_{nextnode} = Fscore_{initial} + \Delta_{small}$$
$$Fscore_{node} = Fscore_{nextnode}$$

Reconstructing the path The method *construct path()* will build an ancestor path to the goal node, so that we are able to find our path to the solution. This would represent the different squares to move a agent in a pathfinding exercise.

The implementation is found on the next page.

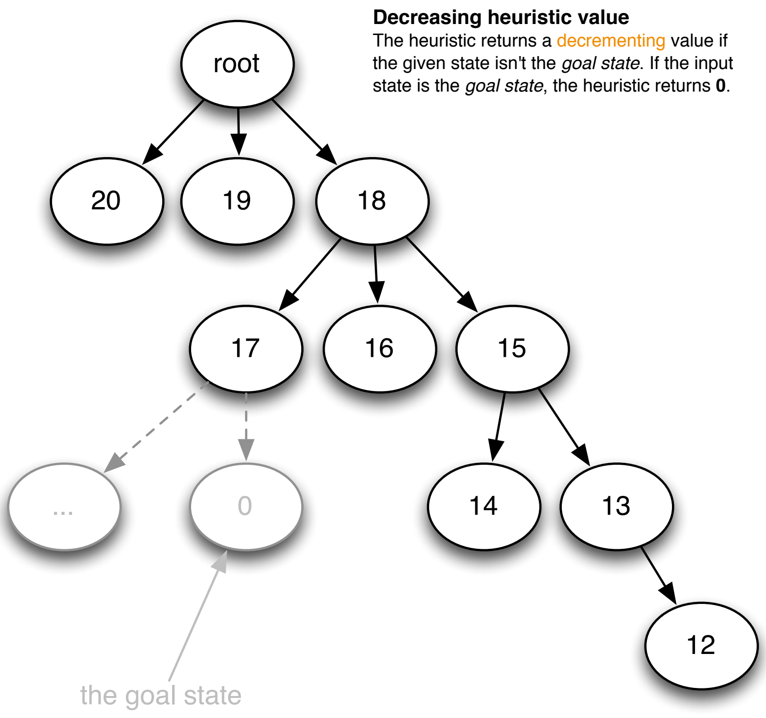
```
from Queue import PriorityQueue
import math
import itertools


exists = {} # Keep track of existing nodes, so that one node may have multiple parents.
            # This is required if you wish to find the optimal path.
exist_key = set() # for a faster existence check.

class AstarNode:
   parent = None
   f_score = None
   h_score = None
   is_expanded = False
   is_observed = False
   g_score = ( )
   children = [ ]

   def generate_children(self):
      def swap( index ):
          seq = list(sequence)
          seq[sep], seq[index] = seq[index], '_'
          return ''.join(seq)
      #end swap
      sequence = self.sequence
      sep = sequence.index('_')
      valid_moves = filter(
                          # filter out the invalid moves such as moving a piece out of the puzzle
                          lambda x: len(sequence)>x>=0,
                          [sep-2,sep-1,sep+1,sep+2]
                          )

      valid_children = list(itertools.imap(swap, valid_moves))
      for seq in valid_children:
          node = Node( seq )
          node.children = []
          if node.key not in exist_key:
             exists.update({node.key:node})
             exist_key.add(node.key)
          self.children.append(exists[node.key])


class Node( AstarNode ):
    def __init__(self, sequence):
        self.sequence = sequence
        self.key = sequence # unique key for hashing purposes.

    def __repr__(self):
       return self.sequence

# "precompiling" the heuristic math function.
# This is especially effective with the PYPY iterpreter.
split_distance = lambda x,y: abs(x-y)
def heuristic(node):
    # goals
    goal_r = xrange(0,(len(node.sequence)-1)/2)
    goal_b = xrange(goal_r[-1]+2, len(node.sequence))
    #

    reds = sorted([index for index,c in enumerate(node.sequence) if c=='r' ])
    blacks = sorted([index for index,c in enumerate(node.sequence) if c=='b' ])

    tot = sum(itertools.imap(split_distance, reds, goal_r))
    tot += sum(itertools.imap(split_distance, blacks, goal_b))
```

```python
    return tot

def distance(parent,child):
   """
   We dont have any definition of the distance between two nodes.
   """
   return 1


def astar(start):
   queued_nodes = PriorityQueue()
   start.g_score = 0
   start.h_score = heuristic(start)
   start.f_score = start.g_score + start.h_score
   start.is_observed = True
   queued_nodes.put((start.f_score,start))

   nn = 0

   while not queued_nodes.empty():
      current = queued_nodes.get()[1] # [f_score, node]
      nn += 1
      if current.h_score == 0:
         return construct_path(current),nn
      else:
         current.is_observed = True
         current.generate_children()
         current.is_expanded = True
         for child in current.children:
            evaled_g_score = current.g_score + distance(current,child)
            if evaled_g_score<child.g_score:
               if child.is_expanded:
                  child.parent = current
                  child.g_score = evaled_g_score
                  child.f_score = child.g_score + child.h_score
                  current.is_expanded = False
                  child.is_observed = False
                  queued_nodes.put((child.f_score,child))
               elif child.is_observed:
                  child.parent = current
                  child.g_score = evaled_g_score
               else:
                  current.is_observed = True
                  child.parent = current
                  child.g_score = evaled_g_score
                  child.h_score = heuristic(child)
                  child.f_score = child.g_score + child.h_score
                  queued_nodes.put((child.f_score,child))
   return Exception('No result')

def construct_path(node):
   tmp = []
   while node.parent:
       tmp.append( node.parent )
       node = node.parent
   return tmp


"""
Example usage:
print astar( start_node )
"""
gene = "bbb_rrr"
out = astar( Node(gene) )
print "nodes:", out[1], "\tsteps:", len(out[0])
```

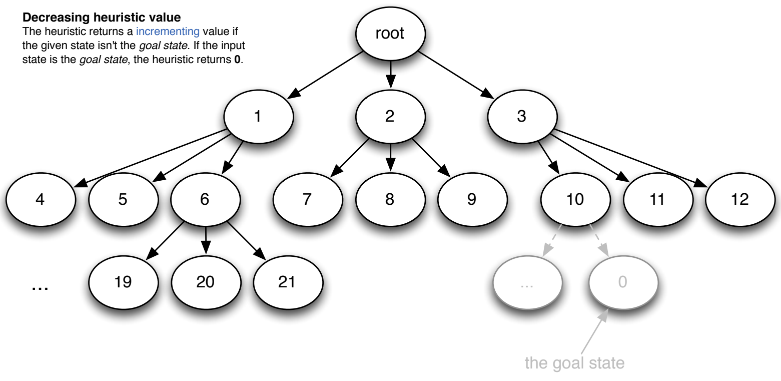
2 Notes to the implementation

The A* implementation will run until a goal state is discovered, and then return the found state. Disregarding possible better solution that were hidden due to a poorly implementet heuristic. Thus the bread-first implementation could in theory be able to discover better solutions since such a thourough search takes care to expand every single "parent node". By running the depth-first version of the search, we are playing a game of chance. We might have to go through tonns of nodes to reach our goal. Since each state has on average 3 possible children and we might have to search deeper than 1000 levels, we would a probability of reaching the optimal state with $< 0.3^{1000} = 1.32 * 10^{-523}$. The heuristics used is fairly good. The script completes a 192-checkers in less than 5 seconds.

How I made a depth first algorithm


Decreasing heuristic value
The heuristic returns a decrementing value if the given state isn't the *goal state*. If the input state is the *goal state*, the heuristic returns **0**.
root
20
19
18
17
16
15
...
0
14
13
12
the goal state

How I made a depth first algorithm


Decreasing heuristic value
The heuristic returns a incrementing value if the given state isn't the *goal state*. If the input state is the *goal state*, the heuristic returns **0**.
root
1
2
3
4
5
6
7
8
9
10
11
12
...
19
20
21
...
0
the goal state

How does it generate children?

Move set The script generates a set of valid moves, which then dictates the child states to be initialized.

```python
def generate_children(self):
  def swap( index ):
      """
      This function swaps the symbol at {index} with the
      separator symbol _
      """
      seq = list(sequence)
      seq[sep], seq[index] = seq[index], '_'
      return ''.join(seq)
  #end swap

  sequence = self.sequence # the symbolic representation of this state. eg: rrr_bbb
  sep = sequence.index('_') # the index of the separator
  valid_moves = filter(
                         # filter out the invalid moves such as moving a piece out of the puzzle
                         lambda x: len(sequence)>x>=0,
                         [sep-2,sep-1,sep+1,sep+2]
                         )
  valid_children = list(itertools.imap(swap, valid_moves))
  for seq in valid_children:
      """
      Loop over the valid children as generated.
      If the child's hash key exists, insert a reference
      to the already existing child.
      Else we create a new entry
      """
      node = Node( seq )
      node.children = []
      if node.key not in exist_key:
         exists.update({node.key:node})
         exist_key.add(node.key)
      self.children.append(exists[node.key])
```

Figure 1: The python child generator

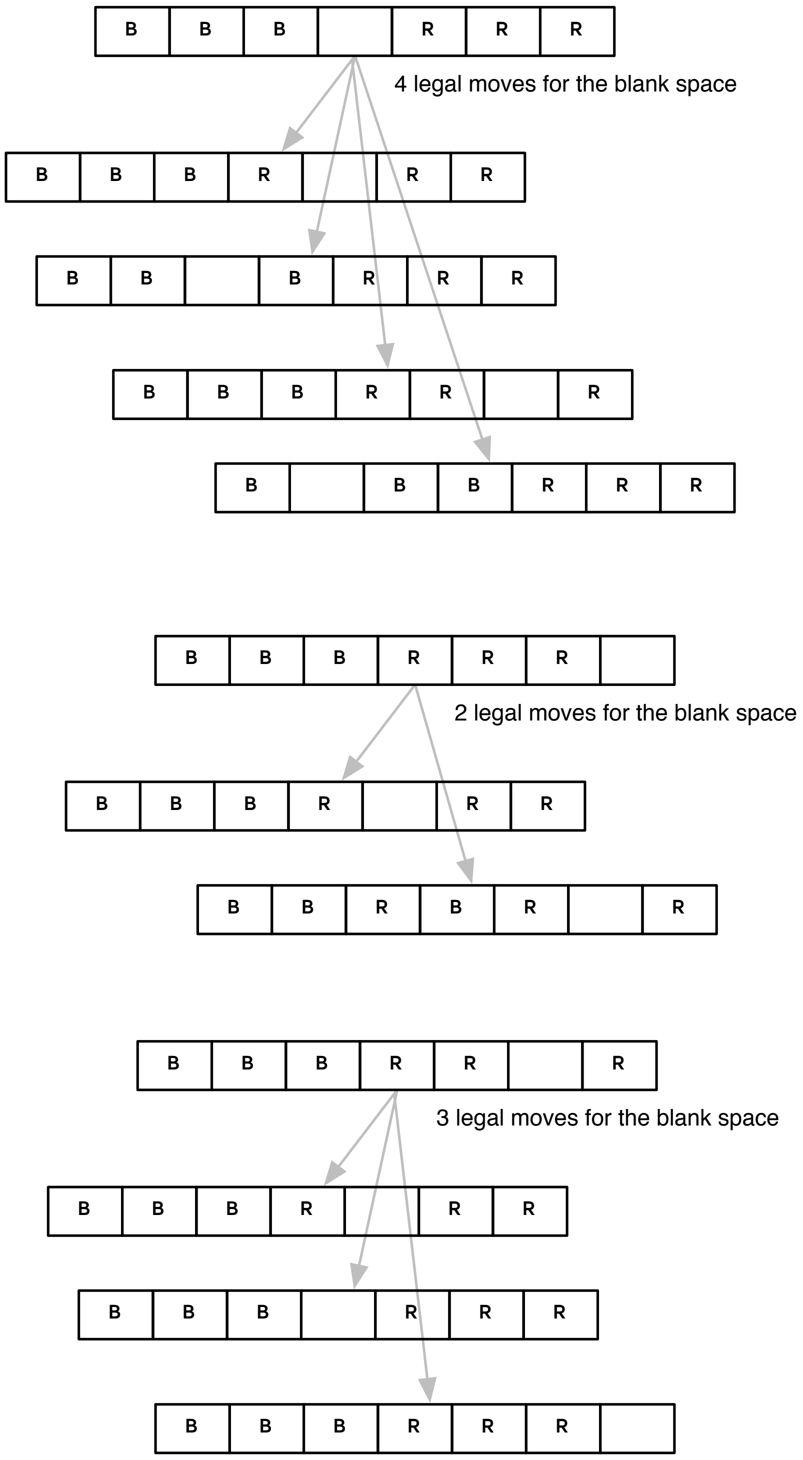


Figure 2: Valid moves in different states.

3 Results

	K-checker			
Search algorithm	6	12	24	30
breadth first	(136,15)	(11992,48)	- >22.369.000 nodes	-
depth first	(88,41)	(4052,2765)	- >10.199.000 nodes	-
A*	(21,15)	(69,48)	(246,168)	(375,255)

An overview description of the solutions to 6-checkers and 30-checkers

6-checkers This is the easiest problem instance to solve. The A* algorithm only has to inspect 21 states to find the optimal path of 15 steps. The implementation could be improves by not generating all the states before they are necessary, but since memory is not an issue I have ignored that approach. The total child nodes generated is 64 nodes.

30-checkers This is the hardest problem instance in this exercise. The algorithm has to inspect a total of 375 states, but generates in fact 1444 child nodes in total. The optimal path consists of 255 steps from the starting state to our goal. The number of generated children is still much lower than what is needed for the bredth-firsth search, which gave up at 22.4 million nodes.

addendum The sequence of generated children respects figure 2.