Missionaries and Cannibal Problem

Ayushi Dixit

September 30, 2022

1 Solution

```
from copy import deepcopy
from collections import deque
import sys
import time


class State(object):
  def __init__(self, missionaries, cannibals , boats):
    self.missionaries = missionaries
    self.cannibals = cannibals
    self.boats = boats

  def successors(self):
    if self.boats == 1:
      sgn = -1
      direction = "from the original shore to the new shore"
    else:
      sgn = 1
      direction = "back from the new shore to the original shore"
    for m in range(3):
      for c in range(3):
        newState = State(self.missionaries+sgn*m, self.cannibals+sgn*c, self.boats+sgn*1);
        if m+c >= 1 and m+c <= 2 and newState.isValid():    # check whether action and resul
          action = "take %d missionaries and %d cannibals %s. %r" % ( m, c, direction, newSt
          yield action, newState

  def isValid(self):
    # first check the obvious
    if self.missionaries < 0 or self.cannibals < 0 or self.missionaries > 3 or self.cannibal
      return False
    # then check whether missionaries outnumbered by cannibals
    if self.cannibals > self.missionaries and self.missionaries > 0:    # more cannibals tha
      return False
```

```
    if self.cannibals < self.missionaries and self.missionaries < 3:    # more cannibals th
      return False
    return True

  def is_goal_state(self):
    return self.cannibals == 0 and self.missionaries == 0 and self.boats == 0

  def __repr__(self):
    return "< State (%d, %d, %d) >" % (self.missionaries, self.cannibals, self.boats)



class Node(object):
  def __init__(self, parent_node, state, action, depth):
    self.parent_node = parent_node
    self.state = state
    self.action = action
    self.depth = depth

  def expand(self):
    for (action, succ_state) in self.state.successors():
      succ_node = Node(
                       parent_node=self,
                       state=succ_state,
                       action=action,
                       depth=self.depth + 1)
      yield succ_node

  def extract_solution(self):
    solution = []
    node = self
    while node.parent_node is not None:
      solution.append(node.action)
      node = node.parent_node
    solution.reverse()
    return solution


def breadth_first_tree_search(initial_state):
  initial_node = Node(
                      parent_node=None,
                      state=initial_state,
                      action=None,
                      depth=0)
  fifo = deque([initial_node])
  num_expansions = 0
```

```
  max_depth = -1
  while True:
    if not fifo:
      print ("%d expansions" % num_expansions)
      return None
    node = fifo.popleft()
    if node.depth > max_depth:
      max_depth = node.depth
      print ("[depth = %d] %.2fs" % (max_depth, time.clock()))
    if node.state.is_goal_state():
      print ("%d expansions" % num_expansions)
      solution = node.extract_solution()
      return solution
    num_expansions += 1
    fifo.extend(node.expand())


def usage():
  print >> sys.stderr, "usage:"
  print >> sys.stderr, "    %s" % sys.argv[0]
  raise SystemExit(2)


def main():
  initial_state = State(3,3,1)
  solution = breadth_first_tree_search(initial_state)
  if solution is None:
    print ("no solution")
  else:
    print ("solution (%d steps):" % len(solution))
    for step in solution:
      print ("%s" % step)
  print ("elapsed time: %.2fs" % time.clock())


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