[CSE547] The Searching Pac-Man

REPORT for TEAM PROJECT #1

by Oleksii Starov, Hyungjoon Koo

Sep. 26, 2013

A. General Information

(1) Team-up (Group 41): Oleksii Starov (ostarov@cs.stonybrook.edu, Main coder, Heuristics), Hyungjoon Koo (hykoo@cs.stonybrook.edu, Search Algorithm, Analysis and Testing)

(2) Project information: Pacman Search (PJT#1), http://www.cs.sunysb.edu/~ram/cse537/project01-2013.html (Due date: Sep. 26, 2013)

(3) Implementation: Q1=DFS, Q2=BFS, Q3=UCS, Q4=A*, Q5-6=SeachAgents finding all corners, Q6=SearchAgent eating all the dots

B. Result Table

Algorithm	Parameters	kinds	Agents	Cost	Running Time(sec)	# of Nodes expanded	Comment
depthFirstSearch	**dfs**	**tinyMaze**	**SearchAgent**	**10**	**0.002**	**14**	
depthFirstSearch	**dfs**	**mediumMaze**	**SearchAgent**	**130**	**0.021**	**144**	
depthFirstSearch	**dfs**	**bigMaze**	**SearchAgent**	**210**	**0.055**	**390**	
breadthFirstSearch	bfs	tinyMaze	SearchAgent	8	0.002	15	
breadthFirstSearch	**bfs**	**mediumMaze**	**SearchAgent**	**68**	**0.035**	**267**	
breadthFirstSearch	**bfs**	**bigMaze**	**SearchAgent**	**210**	**0.092**	**617**	
uniformCostSearch	ucs	tinyMaze	SearchAgent	8	0.022	15	
uniformCostSearch	**ucs**	**mediumMaze**	**SearchAgent**	**68**	**0.040**	**268**	
uniformCostSearch	**ucs**	**bigMaze**	**SearchAgent**	**210**	**0.084**	**619**	
uniformCostSearch	**ucs**	**mediumDottedMaze**	**StayEastSearchAgent**	**1**	**0.023**	**186**	
uniformCostSearch	ucs	mediumDottedMaze	StayWestSearchAgent	17183894840	0.036	169	The pacman can't find the goal for good.
uniformCostSearch	ucs	mediumScaryMaze	StayEastSearchAgent	1	0.028	230	The pacman was killed because it kept going east.
uniformCostSearch	**ucs**	**mediumScaryMaze**	**StayWestSearchAgent**	**68719479864**	**0.016**	**108**	
aStarSearch	**astar,heuristic=manhattanHeuristic**	**bigMaze**	**SearchAgent**	**210**	**0.105**	**585**	
aStarSearch	**astar,heuristic=nullHeuristic**	**bigMaze**	**SearchAgent**	**210**	**0.079**	**619**	
breadthFirstSearch	bfs,prob=CornersProblem	tinyCorners	SearchAgent	28	0.049	409	
breadthFirstSearch	bfs,prob=CornersProblem	mediumCorners	SearchAgent	106	0.806	2380	

[NOTE] The result is based on a certain Macbook. Therefore running time could be relatively different – slow or faster - from different machines.

The ***bold*** illustrates that the result on the project instruction in particular.

The table below shows the test results with what we have implemented. The number of nodes expanded, running time, and cost for each search strategy vary depending on the algorithms (DFS, BFS, UCS, aStar and so forth), the agents, and the size of mazes.

We were able to learn the following from the results:

a. As the size of mazes grows, the cost, running time and the number of nodes expanded increase in general.

b. The time complexity of DFS is **O(b^m)**, where m is the maximum depth of search space, while that of BFS is **O(b^d)**, where d is the depth of the shallowest solution. However, DFS is much better from space complexity perspective, because DFS is **O(bm)** while BFS is still O(**b^d**). (0.021 VS 0.035, 0.055 VS 0.092).

c. In UCS, we can get very low (1) and high (68,719,479,864) path costs for **StayEastSearchAgent, StayWestSearchAgent** because of exponential cost functions.

d. We can see that A* search is slightly faster than UCS (585 VS 619) in the table.

C. Implementation Note

In general, we implemented a common function which returns results, called **getSolution()** which takes three parameters: *theGoal, frontierNodes, trackActions.* This function uses **Stack** to keep track of frontierNode (the pair of the connectivity between adjacent vertexes) and trackActions (keep the pair between vertex and possible action) dictionary. When the search algorithm finds its goal, then this function returns the path in reverse order - from starting point to the goal state.

```
    # Initialize variables to keep track of paths starting from the goal
    PathGoalToStart = util.Stack()
    PathStartToGoal = []
    trackNode = theGoal

    # Track all paths from frontierNodes and trackActions
    # frontierNodes keep the pair of the connectivity between adjacent vertexes.
    # trackActions keep the pair between vertex and possible action
    while trackNode in frontierNodes:
       PathGoalToStart.push(trackActions.get(trackNode)) # Push all items into stack
       trackNode = frontierNodes[trackNode]
    while not PathGoalToStart.isEmpty():
        PathStartToGoal.append(PathGoalToStart.pop()) # Pop all items into solution list

    return PathStartToGoal
```

The following is the **generalErrorDefined()** function in *util.py* when any solution has not been found in the end.

```
def generalErrorDefined():
  print "Search wasn't successful: %s" % inspect.stack()[1][3]
  sys.exit(1)
```

(1) **DFS (Q1)**: DFS (Depth-first search) chooses the shallowest unexpanded node first in the current frontier of the search tree for expansion.

We use ***Stack*** class from *util.py* to properly store successors.

```
# Do iteration until the stack is empty
   while not dfsStack.isEmpty():
      # Pop the node in a stack
      currentNode = dfsStack.pop()
      # Add the node to the exploredNodes list
      exploredNodes.append(currentNode)
      for (nextState, action, cost) in problem.getSuccessors(currentNode):
         # Only if this successor is not in the exploredNodes list
         if nextState not in exploredNodes:
            # Add this successor to the exploredNodes list
            exploredNodes.append(nextState)
            # Save tracking information -
            # from what parent and with what action we came
            trackParents[nextState] = currentNode
            trackActions[nextState] = action
            if problem.isGoalState(nextState):
               # If a goal is found - return this solution
               return getSolution(nextState, trackParents, trackActions)
            # Otherwise push a successor into the stack
            dfsStack.push(nextState)
```

(2) **BFS (Q2)**: BFS (Breadth-first search) chooses the deepest unexpanded node first in the current frontier of the search tree for expansion.

We use ***Queue*** class from *util.py* to properly store successors

The only difference between DFS and BFS in implementation is whether to use Stack or Queue when expanding successors.

```
# Do iteration until the queue is empty
while not bfsQueue.isEmpty():
      # Dequeue the node in a queue
      currentNode = bfsQueue.pop()
      # Add the node to the exploredNodes list
      exploredNodes.append(currentNode)
      for (nextState, action, cost) in problem.getSuccessors(currentNode):
         # Only if this successor is not in the exploredNodes list
         if nextState not in exploredNodes:
            # Add this successor to the exploredNodes list
            exploredNodes.append(nextState)
            # Save tracking information -
            # from what parent and with what action we came
            trackParents[nextState] = currentNode
            trackActions[nextState] = action
            if problem.isGoalState(nextState):
               # If a goal is found - return this solution
               return getSolution(nextState, trackParents, trackActions)
             # Otherwise enqueue a successor into the queue
            bfsQueue.push(nextState)
```

(3) **UCS (Q3)**: UCS (Uniform cost search) expands the node n with the lowest path cost ***g(n),*** which defines the cost to reach the node.

We use ***PriorityQueue*** class from util.py to store the successors. We modified ***PriorityQueue*** class so that it returns both priority and node at the same time.

```
# Do iteration until the queue is empty
while not uscQueue.isEmpty():
       # We retrieve a node with the best priority (cost or distance)
       (priority, node) = uscQueue.pop()
       '''
       This implementation uses the following trick -
       we cannot change priority of an element inside priority queue,
       but we need to do it during the relaxation phase.
       So indeed during relaxation we just add to the queue new values,
       not removing for a vertex its old priority in the queue.
       So we need to check for the following fictive nodes.
       More details can be found in Russian at e-maxx.ru :)
       '''
       # Fictive nodes will be omitted
       if priority > currentDistance[node]:
          # meaning an old one  - let's pop the next
          continue
       if problem.isGoalState(node):
          # If a goal is found - return this solution
          # Note, we should check it here! not while generating children
          # as in previous algorithms
          return getSolution(node, trackParents, trackActions)
       exploredNodes.append(node)
       for (nextState, action, cost) in problem.getSuccessors(node):
          if nextState in exploredNodes:
             # not interesting for us anymore
             continue
          # THE RELAXATION PHASE follows
          if nextState not in currentDistance or priority + cost < currentDistance[nextState]:
             uscQueue.push(nextState, priority + cost)
             currentDistance[nextState] = priority + cost
             # Update tracking information -
             # from what parent and with what action we came
             trackParents[nextState] = node
             trackActions[nextState] = action
```

(4) **A* (Q4)**: A star search involves with heuristic cost, called ***h(n)*** when expanding nodes.

This evaluates every node by combined cost ***f(n) = g(n) + h(n)*** to get from the node to the goal.

The only difference between UCS and A* in implementation is whether to use ***g(n)*** or ***f(n)*** when evaluating the cost for each node.

```
# Do iteration until the queue is empty
while not astarQueue.isEmpty():
       # We retrieve a node with the best priority (cost or distance)
```

```
        (priority, node) = astarQueue.pop()
        # Fictive nodes will be omitted
        if priority > currentDistance[node]:
            # meaning an old one  - let's pop the next
            continue
        if problem.isGoalState(node):
            # If a goal is found - return this solution
            # Note, we should check it here! not while generating children
            # as in previous algorithms
            return getSolution(node, trackParents, trackActions)
        exploredNodes.append(node)
        for (nextState, action, cost) in problem.getSuccessors(node):
            if nextState in exploredNodes:
                # not interesting for us anymore
                continue
            # THE RELAXATION PHASE follows
            if nextState not in currentDistance or priority + cost + heuristic(nextState, problem) < currentDistance[nextState]:
                astarQueue.push(nextState, priority + cost + heuristic(nextState, problem))
                currentDistance[nextState] = priority + cost + heuristic(nextState, problem)
                # Update tracking information -
                # from what parent and with what action we came
                trackParents[nextState] = node
                trackActions[nextState] = action
```

(5) **Finding All Corners (Q5-6)**:

The implemented heuristic function returns the shortest path to visit all previously not visited corners and starting from the current position. To calculate distance between the points on a field the Manhattan distance is used. Hence by this heuristic we are solving a relaxed problem – not taking any walls into account. So it is an admissible one for sure.

It gave us **961 nodes** expanded for "***mediumCorners***" and overall time less than 0.2 seconds.

Optimizations we have tried (commented code):

1) If we have a wall immediately to the right from current position, and we need to go to the right – our Manhattan path will be increased by 2 points at least. Similar thoughts can be applied for each direction including diagonal one (e.g., if we have a wall above and on the right, and need to go to that corner).

2) We can make a good "pre-calculation". We know that Manhattan distance between two corners adjacent to the same side of the map is the length of this side (direct way). But we can find out the highest peak (perpendicular wall) between each pair of such corners. Distance between them will be more or equal to Manhattan distance + 2 * peak's height between them.

Combination of those optimizations gave us the following results: 802 nodes expanded in less than 0.4 seconds. So pre-calculation takes time (especially with raw implementation), but on big maps this variant should be beneficial.

We also tried to use some approaches from http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html but nothing really worked – probably too many walls.

```
class CornersProblem(search.SearchProblem):
  """
  This search problem finds paths through all four corners of a layout.
  You must select a suitable state space and successor function
  """

  def __init__(self, startingGameState):
    """
    Stores the walls, pacman's starting position and corners.
    """
    self.walls = startingGameState.getWalls()

    # Just for second variant of heuristic - precalc of peaks
    self.obs = calcMaxWalls(self.walls)

    self.startingPosition = startingGameState.getPacmanPosition()
    top, right = self.walls.height-2, self.walls.width-2
    self.corners = ((1,1), (1,top), (right, 1), (right, top))
    for corner in self.corners:
      if not startingGameState.hasFood(*corner):
        print 'Warning: no food in corner ' + str(corner)
    self._expanded = 0 # Number of search nodes expanded

    '''
    Our state representation includes position tracking
    (it's necessary for successors generation),
    and a list of visited vertices
    '''
    visitedCorners = []
    # If current position is already a corner - check
    if self.startingPosition in self.corners:
        visitedCorners.append(self.startingPosition)

    # This is more a universal code to keep list of visited vertices,
    # e.g., if not only corners will be needed - it will work.
    # More efficient specific solution would be a Bit Mask
    self.startState = (self.startingPosition, tuple(visitedCorners))

  def getStartState(self):
    "Returns the start state (in your state space, not the full Pacman state space)"
    # THIS IS A NEW CODE BY OLEKSII & HYUNGJOON
    # We will just return our state
    return self.startState

  def isGoalState(self, state):
    "Returns whether this search state is a goal state of the problem"
    # THIS IS A NEW CODE BY OLEKSII & HYUNGJOON
    # The indicator of the goal reached is if all vertices (corners) are visited!
    return len(list(state[1])) == len(self.corners)

  def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.

     As noted in search.py:
         For a given state, this should return a list of triples,
     (successor, action, stepCost), where 'successor' is a
     successor to the current state, 'action' is the action
     required to get there, and 'stepCost' is the incremental
     cost of expanding to that successor
    """
    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
      # Add a successor state to the successor list if the action is legal
      # Here's a code snippet for figuring out whether a new position hits a wall:
      #   x,y = currentPosition
      #   dx, dy = Actions.directionToVector(action)
      #   nextx, nexty = int(x + dx), int(y + dy)
      #   hitsWall = self.walls[nextx][nexty]

      # Similar successors generation but with proper visitedCorners list propagation!
      x, y = state[0]
      dx, dy = Actions.directionToVector(action)
      nextx, nexty = int(x + dx), int(y + dy)
      if not self.walls[nextx][nexty]:
        # Copy the parent's visited list to a child (second half of a tuple)
        updatedVisited = list(state[1])
        # If chile is a new corner - remove it from it's visited list
        if (nextx, nexty) in self.corners and (nextx, nexty) not in updatedVisited:
            updatedVisited.append((nextx, nexty))
        # We've get a next state specified
        nextState = ((nextx, nexty), tuple(updatedVisited))
        successors.append((nextState, action, 1))

    # supporting stuff
    self._expanded += 1
    return successors
```

(6) Eating All The Dots Heuristic (Q7):

The implemented heuristic function is very simple – it just returns length of the list of not yet visited points with food. The goal state will have it 0. It's obvious that this heuristic is admissible, because overestimation is just not possible – what can be less than the straight chain of all food? It provides the most optimistic estimation. At the same time it is consistent, because with each step that costs 1 our estimation can stay the same (if it wasn't a food cell), or decrease only on 1 (which is the same with a coast of the move). So the all

Without heuristics, it takes 16,469 nodes expanded in more than 62 seconds.

Possible optimization (commented code) – We tried to count gaps between food-dots that for sure will be along the line path (gives improvement up to **800** expanded nodes less). Here we made an attempt to leverage "walls" information.

Other heuristic (commented code) – MANHATTAN PATH BY CONVEX HULL.

requirements of consistency met:

$$h(N) \leq c(N,P) + h(P)$$
$$h(G) = 0.$$

Results: **8,679 nodes** expanded, 15.5 seconds (path with cost of 60).

The code below contains a couple of approaches to solve the problem.

In short, the idea is to calculate convex of food-dots, and to remove the maximal segment (because our current position may not use it). It won't overestimate the correct path, so it is admissible. But in the current implementation it is not consistent. Some workaround is needed – something like trying to include current position to the hull. But results are very cool – **3,857** nodes expanded in 4.35 seconds. But the path cost became 68 instead of 60 (slightly more).

```
class AStarFoodSearchAgent(SearchAgent):
  "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
  def __init__(self):
    self.searchFunction = lambda prob: search.aStarSearch(prob, foodHeuristic)
    self.searchType = FoodSearchProblem

def foodHeuristic(state, problem):

  position, foodGrid = state

  '''
  This HEURISTIC is for sure admissible and consistent.
  The relaxed representation operates with the smallest possible path -
  if all left dots are accessible in a line from the current position
  So we can't get an overestimation. At the same time - next state
  can be 1 dot less or just equal - consistency is kept.
  RESULTS: 8679 nodes
  '''

  # Just so simple :)
  return len(foodGrid.asList())

  '''
  # POSSIBLE OPTIMIZATION OF THE PREVIOUS IDEA
  # trying to calculate number of gaps in a line of left dots
  # that will be for sure
  res = len(foodGrid.asList())

  cnt = 0
  for (x, y) in foodGrid.asList():
     #if isAlone(x, y, foodGrid):
     cnt = cnt + isAlone(x, y, foodGrid)

  return res + cnt/2
  '''

  '''
  # ANOTHER APPROACH - MANHATTAN PATH ALONG CONVEX HULL
  # Admissible but possible inconsistent heuristic,
  # some modifications needed to make it consistent,
  # like taking into account a current position

  l = foodGrid.asList()
  #l.append(position)
  hull = convex_hull(l)

  #print (position)
  #print (hull)
  #print (">")
  #print ()

  #ind = hull.index(position)

  # Base cases
  if len(hull) == 0:
     return 0
  if len(hull) == 1:
     return 1
  if len(hull) == 2:
     return abs(hull[0][0] - hull[1][0]) + abs(hull[0][1] - hull[1][0])

  res = 0
  maxDist = -1
  hull.append(hull[0])
  for i in range(0, len(hull)-1):
     # Again using manhattan distances
     dist = abs(hull[i][0] - hull[i+1][0]) + abs(hull[i][1] - hull[i+1][0])
     if maxDist < dist:
        # Removing the longest segment, because p could be closer
        maxDist = dist
     res = res + dist
  res = res - maxDist

  return res
  '''

def isAlone(x, y, foodGrid):
   # Improvement for min line heuristic
   if foodGrid[x+1][y] or foodGrid[x][y+1]:
      return 0
   if foodGrid[x-1][y] or foodGrid[x][y-1]:
      return 0
   return 1

# http://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain
def convex_hull(points):
   """
   Computes the convex hull of a set of 2D points.
   Input: an iterable sequence of (x, y) pairs representing the points.
   Output: a list of vertices of the convex hull in counter-clockwise order,
     starting from the vertex with the lexicographically smallest coordinates.
```

```
    Implements Andrew's monotone chain algorithm. O(n log n) complexity.
    """

    # Sort the points lexicographically (tuples are compared lexicographically).
    # Remove duplicates to detect the case we have just one unique point.
    points = sorted(set(points))

    # Boring case: no points or a single point, possibly repeated multiple times.
    if len(points) <= 1:
        return points

    # 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product.
    # Returns a positive value, if OAB makes a counter-clockwise turn,
    # negative for clockwise turn, and zero if the points are collinear.
    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    # Build lower hull
    lower = []
    for p in points:
        while len(lower) >= 2 and cross(lower[-2], lower[-1], p) <= 0:
            lower.pop()
        lower.append(p)

    # Build upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    # Concatenation of the lower and upper hulls gives the convex hull.
    # Last point of each list is omitted because it is repeated at the beginning of the other list.
    return lower[:-1] + upper[:-1]
```