به نام خدا

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

مبانی و کاربردهای هوش مصنوعی (پاییز ۱۴۰۱)

گزارش پروژه - فاز اول

استاد درس:

دکتر جوانمردی

نام دانشجو:

محمدجواد رضوانیان

مهلت ارسال تمرین: ۱۱ آذر ماه ۱۴۰۱

پاسخ سوال ۰:

این کلاس یک طرح کلی از مسئله‌ی جست و جو را نشان می‌دهد.

این کلاس شامل چهار متد است؛

```
getStartState(self)
isGoalState(self, state)
getSuccessors(self, state)
getCostOfActions(self, actions)
```

به ترتیب به توضیح هر کدام می‌پردازیم:

- متد اول وظیفه‌ی این را دارد که حالت ابتدایی مسئله را برگرداند.

```
Returns the start state for the search problem.
```

- متد دوم وظیفه این را دارد که با گرفتن حالت فعلی مسئله جست و جو، چک کند که آیا وضعیتی که در آن قرار گرفته ایم همان وضعیت مطلوب و نهایی ما است یا خیر.

```
Returns True if and only if the state is a valid goal state.
```

- متد سوم وظیفه‌ی این را دارد که با گرفتن وضعیت فعلی، مقادیر سه گانه‌ی زیر را مشخص کند؛ successor، action و stepcost. Successor مشخص می‌کند که می‌تواند به کدام وضعیت‌ها به وسیله این وضعیت برسد. Action مشخص می‌کند که با کدام اقداماتی می‌توان به وضعیت‌های مشخص شده رسید و در نهایت هم stepcost مشخص می‌کند که هزینه انجام هر کدام از این اقدامات برای رسیدن به وضعیت جدید چه قدر است.

```
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.
```

- متد چهارم، مقدار هزینه مجموعه اقدامات را برمیگرداند. مجموعه اقداماتی که امکان انجام آن وجود دارد.

```
This method returns the total cost of a particular sequence of actions.
The sequence must be composed of legal moves.
```

پاسخ سوال ۰:

کلاس Agent:

- این کلاس دارای دو متد است که متد اول نقش سازنده را دارد و متد دوم وظیفه این را دارد که وضعیت را از فایل‌های مناسب خود گرفته و اقدام مناسب را انتخاب کند.

```
The Agent will receive a GameState (from either {pacman, capture, sonar}.py)
and must return an action from Directions. {North, South, East, West, Stop}
```

کلاس Direction:

- این کلاس باید جهات ثانویه را برای اقدام مناسب مشخص کند؛ یعنی اگر وضعیت فعلی در نقطه‌ی x و y و روبه‌سمت مثلا N است، با اقدام چرخش به راست، جهت کاراکتر به سمت E شود.

کلاس Configuration:

- وظیفه اصلی این کلاس، نگهداری موقعیت و جهت عامل ما است. در واقع یکی از وظایف این کلاس این است که عملیات کپسوله سازی را برای ما انجام می‌دهد.

کلاس AgentState:

- این کلاس وظیفه نگهداری وضعیت عامل را دارد.

کلاس Grid:

- وظیفه این کلاس این است که موقعیت عامل ما را به صورت یک آرایه دو بعدی در ساختار grid[x][y] نگهدارد. که نقطه‌ی (۰،۰) گوشه‌ی پایین سمت چپ صفحه است.

Screenshot:

```
def update(self, item, priority):
    # If item already in priority queue with higher priority, update its priority and rebuild the heap.
    # If item already in priority queue with equal or lower priority, do nothing.
    # If item not in priority queue, do the same thing as self.push.

    "*** YOUR CODE HERE ***"
    # (Index, (Priority, Object content, Item))
    for index, (p, c, i) in enumerate(self.heap):
        if i == item:
            if p <= priority:
                # Not do it any action and must be exit from loop because this item priority is lower.
                break
            # Delete the item from priority queue and append new item to it.
            del self.heap[index]
            # Append item to heap DS.
            self.heap.append((priority, c, item))
            # Heapify the DS.
            heapq.heapify(self.heap)
            break
    # Push the last item onto the priority queue if not exist.
    else:
        self.push(item, priority)
```

پاسخ سوال ۱:

الگوریتم جست و جوی عمیق تکراری (Iterative Deepening Search) یا جست‌وجوی عمق اول تکرار شونده (IDDFS)، یک ترکیبی از الگوریتم های DFS و BFS است. این الگوریتم برمبنای جست‌وجوی عمق اول اما با رویکرد محدودیت در عمق کار می‌کند. در ابتدا یک عمق را مشخص می‌کند و تا آن عمق با استفاده از الگوریتم DFS جست‌وجو را انجام می‌دهد.

این الگوریتم بدلیل بازگشتی بودن اینگونه عمل می‌کند که خط اول به اندازه‌ی max_depth بار بررسی می‌شود و آخرین برگ درخت به اندازه ۱ بار بررسی می‌شود.

```
// Returns true if target is reachable from
// src within max_depth
bool IDDFS(src, target, max_depth)
    for limit from 0 to max_depth
       if DLS(src, target, limit) == true
           return true
    return false

bool DLS(src, target, limit)
    if (src == target)
        return true;

    // If reached the maximum depth,
    // stop recursing.
    if (limit <= 0)
        return false;

    foreach adjacent i of src
        if DLS(i, target, limit?1)
            return true

    return false
```

برای تبدیل الگوریتم DFS به IDS کافی است که برای الگوریتم عمق اول یک محدودیت در درخت جست‌وجو قرار دهیم و اجازه ندهیم که تا آخرین برگ را پیمایش کند و این کار را با عمق ۱ شروع کنیم و با هر بار جست و جو مقدار max_depth را هر بار یک واحد اضافه کنیم.

Screenshot:

```python
def fringeInitializer(problem, DS):
    fringe = DS
    # Location of the agent for isGoalState checking in follow.
    startLocation = problem.getStartState()

    if type(DS) == type(util.PriorityQueue()):
        # (location, path, cost)
        startNode = (startLocation, [], 0)
        fringe.push(startNode, 0)
    else:
        startNode = (startLocation, [])
        fringe.push(startNode)
    # Set of nodes in the queue to be searched
    # {visitedLocation, (True, False)}
    visitedLocation = set()
    return fringe, visitedLocation, startLocation;

def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"

    fringe, visitedLocation, startLocation = fringeInitializer(problem=problem, DS=util.Stack())

    while not fringe.isEmpty():
        # node[0] is location, while node[1] is path
        node = fringe.pop()
        visitedLocation.add(node[0])
        if problem.isGoalState(node[0]):
            return node[1]
        successors = problem.getSuccessors(node[0])
        for item in successors:
            if item[0] in visitedLocation:
                continue
            fringe.push((item[0], node[1] + [item[1]]))

    return None
```

DFS

پاسخ سوال ۲:

این الگوریتم در واقع عملیات سطح اول دوطرفه است که برای حل مسئله ۸ پازل استفاده می‌شود و بهینه‌تر از جست‌وجوی سطح اول کار می‌کند. در واقع این راه حل برای زمانی کاربرد دارد که امکان شروع حل مسئله از ۲ جهت وجود دارد (مثل مسئله ۸ پازل که برای یک خانه خالی، از ۲ تا ۴ کاندید برای تغییر حالت وجود دارد)

این الگوریتم با رویکرد روبه جلو (از نود شروع تا نود هدف) و روبه عقب (از نود هدف تا نود شروع) میتواند کار کند. اگر دو گرافی که بررسی می‌شود دارای نقطه اشتراک باشند (بهم برخورد کنند) عملا بررسی گراف متوقف می‌شود.

این الگوریتم هم می‌تواند دارای هیوریستیک باشد.

```
struct Graph {
    // number of nodes in graph
   int V;
    // Adjacency list
   LinkedList<Integer>[] adj;
}
    // Method for adding undirected edge
    void addEdge(int u, int v)
    {
        adj[u].add(v);
        adj[v].add(u);
    }

    // Method for Breadth First Search
    void BFS();

    // check for intersecting vertex
    Boolean is Intersecting(Boolean[] s_visited, Boolean[] t_visited)
    {
        for (int i = 0; i < V; i++) {
            if (s_visited[i] && t_visited[i])
                return i;
        }
        return -1;
    }

    // Method for bidirectional searching
    Boolean biDirSearch(int s, int t)
    {
        // Booleanean array for BFS started from
        // source and target(front and backward BFS)
        // for keeping track on visited nodes
        Boolean[] s_visited = new Boolean[V];
        Boolean[] t_visited = new Boolean[V];

        // Keep track on parents of nodes
        // for front and backward search
        int[] s_parent = new int[V];
        int[] t_parent = new int[V];
```

```
        // queue for front and backward search
        Queue<Integer> s_queue = new LinkedList<Integer>();
        Queue<Integer> t_queue = new LinkedList<Integer>();

        int intersectNode = -1;

        // necessary initialization
        for (int i = 0; i < V; i++) {
            s_visited[i] = false;
            t_visited[i] = false;
        }

        s_queue.add(s);
        s_visited[s] = true;

        // parent of source is set to -1
        s_parent[s] = -1;

        t_queue.add(t);
        t_visited[t] = true;

        // parent of target is set to -1
        t_parent[t] = -1;

        while (!s_queue.isEmpty() && !t_queue.isEmpty()) {
            // Do BFS from source and target vertices
            bfs(s_queue, s_visited, s_parent);
            bfs(t_queue, t_visited, t_parent);

            // check for intersecting vertex
            intersectNode = isIntersecting(s_visited, t_visited);

            // If intersecting vertex is found
            // that means there exist a path
            if (intersectNode != False) {
                System.out.printf("Path exist between %d and %d\n", s, t);
                System.out.printf("Intersection at: %d\n", intersectNode);

                // print the path and exit the program
                printPath();
                return True;
            }
        }
        return False;
    }
}
```

حالا اگر بیش از ۱ هدف داشتیم، می‌توانیم مسیری را انتخاب کنیم که کمترین هزینه برای رسیدن به هر یک از اهداف را دارد انتخاب کنیم. (ممکن است که هزینه برای ما زمان باشد)

Screenshot:

```python
def fringeInitializer(problem, DS):
    fringe = DS
    # Location of the agent for isGoalState checking in follow.
    startLocation = problem.getStartState()

    if type(DS) == type(util.PriorityQueue()):
        # (location, path, cost)
        startNode = (startLocation, [], 0)
        fringe.push(startNode, 0)
    else:
        startNode = (startLocation, [])
        fringe.push(startNode)
    # Set of nodes in the queue to be searched
    # {visitedLocation, (True, False)}
    visitedLocation = set()
    return fringe, visitedLocation, startLocation;

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""

    "*** YOUR CODE HERE ***"
    fringe, visitedLocation, startLocation = fringeInitializer(problem=problem, DS=util.Queue())
    visitedLocation.add(startLocation)

    while not fringe.isEmpty():
        # node[0] is location, while node[1] is path.
        node = fringe.pop()
        if problem.isGoalState(node[0]):
            return node[1]
        # problem.getSuccessors return: (successor, action, stepCost)
        successors = problem.getSuccessors(node[0])
        # List of successors such that [R, L, Reverse]
        for item in successors:
            if item[0] in visitedLocation:
                continue
            # Push object(successor, action)
            fringe.push((item[0], node[1] + [item[1]]))
            # Not visited yet, so be visited first.
            visitedLocation.add(item[0])

    return None
```

پاسخ سوال ۳:

اگر که هزینه‌ی هر انتقال برابر با ۱ باشد، می‌توان گفت که UCS شکل دیگری از BFS است اما این امکان وجود ندارد که UCS به DFS تبدیل شود.

Screenshot:

```python
def fringeInitializer(problem, DS):
    fringe = DS
    # Location of the agent for isGoalState checking in follow.
    startLocation = problem.getStartState()

    if type(DS) == type(util.PriorityQueue()):
        # (location, path, cost)
        startNode = (startLocation, [], 0)
        fringe.push(startNode, 0)
    else:
        startNode = (startLocation, [])
        fringe.push(startNode)
    # Set of nodes in the queue to be searched
    # {visitedLocation, (True, False)}
    visitedLocation = set()
    return fringe, visitedLocation, startLocation;

def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    fringe, visitedLocation, startLocation = fringeInitializer(problem=problem, DS=util.PriorityQueue())

    while not fringe.isEmpty():
        # node[0] is location, while node[1] is path, while node[2] is cumulative cost
        node = fringe.pop()
        if problem.isGoalState(node[0]):
            return node[1]
        if node[0] not in visitedLocation:
            visitedLocation.add(node[0])
            for successor in problem.getSuccessors(node[0]):
                if successor[0] not in visitedLocation:
                    cost = node[2] + successor[2]
                    fringe.push((successor[0], node[1] + [successor[1]], cost), cost)

    return None
```

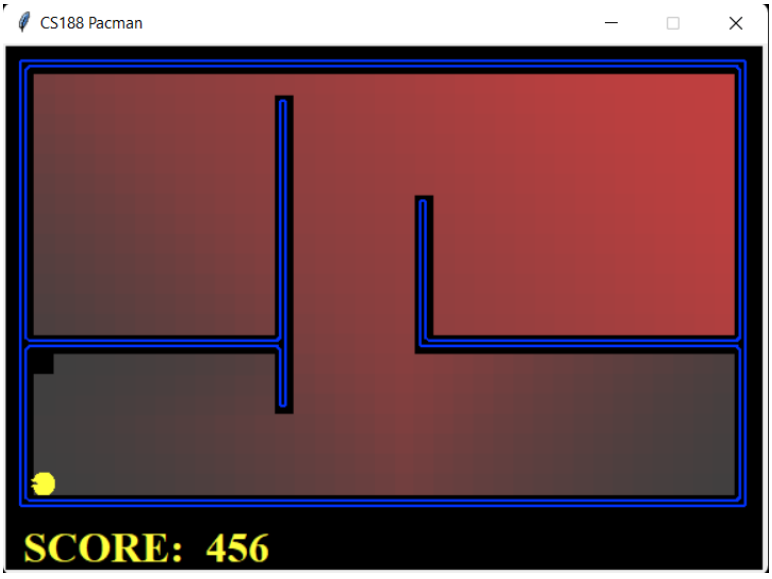
پاسخ سوال ۴:

وقتی که الگوریتم A* بدست آمده را روی نقشه‌ی openMaze امتحان می‌کنیم به نتیجه زیر میرسیم:

گزارش عملکرد عامل به صورت زیر است:

```
P1 python pacman.py -l openMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 535
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

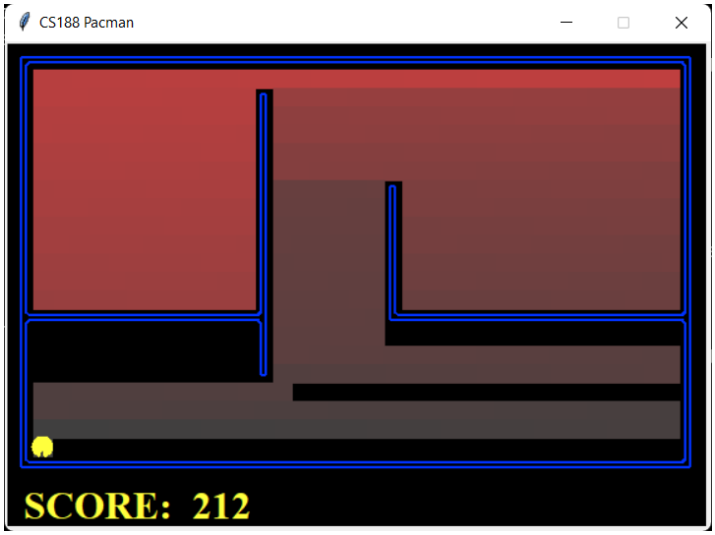
حالا از الگوریتم BFS استفاده می‌کنیم:



گزارش عملکرد به صورت زیر است:

```
P1 python pacman.py -l openMaze -z .5 -p SearchAgent -a fn=bfs,heuristic=manhattanHeuristic
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 54 in 0.0 seconds
Search nodes expanded: 682
Pacman emerges victorious! Score: 456
Average Score: 456.0
Scores:        456.0
Win Rate:      1/1 (1.00)
Record:        Win
```

حالا اگر که از الگوریتم DFS استفاده کنیم، نتیجه به صورت زیر است:



گزارش بدست آمده به صورت زیر است:

```
P1 python pacman.py -l openMaze -z .5 -p SearchAgent -a fn=dfs,heuristic=manhattanHeuristic
[SearchAgent] using function dfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 298 in 0.0 seconds
Search nodes expanded: 806
Pacman emerges victorious! Score: 212
Average Score: 212.0
Scores:        212.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Screenshot:

```python
def manhattanHeuristic(position, problem, info={}):
    "The Manhattan distance heuristic for a PositionSearchProblem"

    "*** YOUR CODE HERE ***"
    p1, p2 = p(position, problem)
    return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

def euclideanHeuristic(position, problem, info={}):
    "The Euclidean distance heuristic for a PositionSearchProblem"

    "*** YOUR CODE HERE ***"
    p1, p2 = p(position, problem)
    return abs( (p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2 ) ** 0.5
```

```python
def fringeInitializer(problem, DS):
    fringe = DS
    # Location of the agent for isGoalState checking in follow.
    startLocation = problem.getStartState()

    if type(DS) == type(util.PriorityQueue()):
        # (location, path, cost)
        startNode = (startLocation, [], 0)
        fringe.push(startNode, 0)
    else:
        startNode = (startLocation, [])
        fringe.push(startNode)
    # Set of nodes in the queue to be searched
    # {visitedLocation, (True, False)}
    visitedLocation = set()
    return fringe, visitedLocation, startLocation;
```

```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    fringe, visitedLocation, startLocation = fringeInitializer(problem=problem, DS=util.PriorityQueue())

    while not fringe.isEmpty():
        # node[0] is location, while node[1] is path, while node[2] is cumulative cost
        node = fringe.pop()
        if problem.isGoalState(node[0]):
            return node[1]
        if node[0] not in visitedLocation:
            visitedLocation.add(node[0])
            for successor in problem.getSuccessors(node[0]):
                if successor[0] not in visitedLocation:
                    cost = node[2] + successor[2]
                    # This is Diffrent between UCS and A*:
                    totalCost = cost + heuristic(successor[0], problem)
                    fringe.push((successor[0], node[1] + [successor[1]], cost), totalCost)

    return None
```

پاسخ سوال ۵:

```
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):···

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        return (self.startingPosition, (0, 1, 2, 3))

        util.raiseNotDefined()

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        return not state[1]

        util.raiseNotDefined()
def getSuccessors(self, state):
    """···

    successors = []
    for action in [Directions.NORTH, Directions.WEST, Directions.SOUTH, Directions.EAST]:
        # 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]

        "*** YOUR CODE HERE ***"
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextX, nextY = int(x + dx), int(y + dy)

        if not self.walls[nextX][nextY]:
            # Change state[1] if reaches corner
            remainedCorners = state[1]
            nextLocation = (nextX, nextY)
            try:
                # Find out if the successor is a corner
                idx = self.corners.index(nextLocation)
            except:
                pass
            else:
                if idx in remainedCorners:
                    temp = list(remainedCorners)
                    temp.remove(idx)
                    remainedCorners = tuple(temp)

            nextState = (nextLocation, remainedCorners)
            successors.append((nextState, action, 1))
```

پاسخ سوال ۶:
(راستش تو اینترنت سرچ کردم، ۶ یا ۷ مدل پیاده‌سازی این قسمت پروژه رو دیدم، دیدم که این روش پیاده‌سازی بهترین جواب را برمی‌گرداند. دقیقا نمی‌توانم با استدلال ثابت کنم.)

Screenshot:

```python
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound on the
    shortest path from the state to a goal of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    "*** YOUR CODE HERE ***"
    p = state[0]
    cr = state[1]

    if cr:
        # max manhattan distance among all remained corners
        return max([util.manhattanDistance(p, corners[idx]) for idx in cr])
    else:
        return 0 # Default to trivial solution
```

گزارش الگوریتم بالا به صورت زیر است:

```
P1 python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 1136
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

حالا اگر به جای فاصله‌ی منهتن، از فاصله اقلیدسی استفاده کنیم به صورت زیر خواهد بود:

```
P1 python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.0 seconds
Search nodes expanded: 1241
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

همانطور که می‌بینیم، تعداد نودهای باز شده در فاصله منهتن برابر با ۱۱۳۶ و در فاصله اقلیدسی برابر با ۱۲۴۱ است.

پاسخ سوال ۷:
(راستش تو اینترنت سرچ کردم، ۶ یا ۷ مدل پیاده‌سازی این قسمت پروژه رو دیدم، دیدم که این روش پیاده‌سازی بهترین جواب را برمی‌گرداند. دقیقا نمی‌توانم با استدلال ثابت کنم.)

Screenshot:

نتیجه استفاده پیاده شدن هیورستیک ارائه شده:

```
P1 python pacman.py -l trickySearch -p AStarFoodSearchAgent
Path found with total cost of 60 in 0.5 seconds
Search nodes expanded: 4137
Pacman emerges victorious! Score: 570
Average Score: 570.0
Scores:        570.0
Win Rate:      1/1 (1.00)
Record:        Win
```

عامل ما نتوانست که مسئله را در mediumSearh حل کند! (لبتاب گیر کرد!)

پاسخ سوال ۸:
این رو حل نکردم.