23 October 2021

Artificial Intelligence

Laboratory activity

Name:Cioara Iulia Maria
Group:30231
Email:cioara_iulia@yahoo.com

Teaching Assistant: Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Chapter 1

A1: Search

Implementarea algoritmilor de cautare in cazul Pacman, urmeaza sa ii detaliez in continuare pe fiecare in parte.

1. (Q1) Depth-first search:

   Acest algoritm pleaca dintr-un nod radacina si gaseste toti succesorii acestuia urmand sa aleaga unul dintre ei. Am folosit o structura pentru a salva nodurile deja vizitate si o lista pentru a adauga nodurile.

```
    def depthFirstSearch(problem):
        visited = []
        start_node = problem.getStartState()
        if problem.isGoalState(start_node):
            return []
        my_stack = util.Stack()
        my_stack.push((start_node, []))
        while not my_stack.isEmpty():
      curr_node, actions = my_stack.pop()
      if curr_node not in visited:
        visited.append(curr_node)
        if problem.isGoalState(curr_node):
          return actions
          for succesor,next_action,
          cost in problem.getSuccessors(curr_node):
            new_action = actions + [next_action]
       my_stack.push((succesor, new_action))

```

2. (Q2) Breadth-first search:

   Acest algoritm porneste dintr-un nod radacina si viziteaza prima data nodurile vecine acestuia inainte de a trece la nivelul urmator. Am folosit o structura pentru a salva nodurile deja vizitate si o coada pentru a adauga nodurile.

```
    def breadthFirstSearch(problem):
             visited = []
        start_node = problem.getStartState()
         if problem.isGoalState(start_node):
            return []
                my_queue = util.Queue()
                my_queue.push((start_node, []))
                while not my_queue.isEmpty():
       curr_node, actions = my_queue.pop()
       if curr_node not in visited:
        visited.append(curr_node)
        if problem.isGoalState(curr_node):
```

```
            return actions
            for succesor, next_action, cost in problem.getSuccessors(curr_node):
             new_action = actions + [next_action]
             my_queue.push((succesor, new_action))

```

3. (Q3) Uniform-cost search:

   Acest algoritm este similar cu BFS si DFS doar ca foloseste cel mai mic cost cumulat pentru a gasi un drum de la sursa la destinatie. Ca structura am folosit o coada de prioritati.

```
def uniformCostSearch(problem):
    visited = []
    start_node = problem.getStartState()
    if problem.isGoalState(start_node):
        return []
    my_priorityQ = util.PriorityQueue()
    my_priorityQ.push((start_node, [], 0), 0)
    while not my_priorityQ.isEmpty():
        curr_node, actions, cost = my_priorityQ.pop()
        if curr_node not in visited:
            visited.append(curr_node)
            if problem.isGoalState(curr_node):
                return actions
            for successor, next_action, next_cost in problem.getSuccessors(curr_node):
                new_action = actions + [next_action]
                priority = cost + next_cost
                my_priorityQ.push((successor, new_action, priority), priority)

```

4. (Q4) A* search:

   Acest algoritm este similar cu uniform cost search, diferenta fiind ca acesta isi calculeaza si costul pentru a ajunge la tinta. Functia pe care o foloseste este f(n)=g(n)+h(n).

```
def aStarSearch(problem, heuristic=nullHeuristic):
    visited = []
    start_node = problem.getStartState()
    if problem.isGoalState(start_node):
        return []
    my_priorityQ = util.PriorityQueue()
    my_priorityQ.push((start_node, [], 0), 0)
    while not my_priorityQ.isEmpty():
        curr_node, actions, p_cost = my_priorityQ.pop()
        if curr_node not in visited:
            visited.append(curr_node)
            if problem.isGoalState(curr_node):
                return actions
            for next, action, cost in problem.getSuccessors(curr_node):
                new_action = actions + [action]
                new_cost = p_cost + cost
                new_heuristic = new_cost + heuristic(next, problem)
                my_priorityQ.push((next, new_action, new_cost), new_heuristic)

```

5. (Q5) Greedy:

   Acest algoritm este o combinatie intre BFS si A* doar ca foloseste o functie heuristica diferita. Calculeaza la nivel local cea mai optima alegere pentru a gasi un optim global.

```
def greedy(problem, heuristic=nullHeuristic):
    start_node = problem.getStartState()
    visited = []
    if problem.isGoalState(start_node):
        return []
    my_priorityQ = util.PriorityQueue()
    my_priorityQ.push((start_node, [], 0), 0)
    while not my_priorityQ.isEmpty():
        curr_node, actions, p_cost = my_priorityQ.pop()
        if curr_node not in visited:
            visited.append(curr_node)
            if problem.isGoalState(curr_node):
                return actions
            for next, action, cost in problem.getSuccessors(curr_node):
                new_action = actions + [action]
                new_cost = p_cost + cost
                my_priorityQ.push((next, new_action, new_cost), heuristic(next, problem))

```

6. (Q6) Iterative Deepening Search:

   Acest algoritm rezuma procesul de cautare intr-o singura functie prin executarea consecutiva a primelor cautari in adancime la niveluri de adancime din ce in ce mai mari, marcate ca si deep. Aici verificam adancimea pe care am atins-o in succesorii nodului curent intr-o bucla.

```
def iterativeDeepeningSearch(problem):
    start_node = problem.getStartState()
    my_stack = util.Stack()
    my_stack.push((start_node, [], 0))
    deep= 0
    while not my_stack.isEmpty():
        deep += 1
        curr_node, actions, cost = my_stack.pop()
        visited = []
        visited.append(curr_node)
        while True:
            for succesor, next_action, next_cost in problem.getSuccessors(curr_node):
                if succesor not in visited and (cost + next_cost) <= deep:
                    visited.append(succesor)
                    my_stack.push((succesor, actions + [next_action], cost + next_cost))
            if my_stack.isEmpty():
                break
            curr_node, actions, cost = my_stack.pop()
            if problem.isGoalState(curr_node):
                return actions
        my_stack.push((start_node, [], 0))

```

Chapter 2

A2: Logics

Chapter 3

A3: Planning