Documentation

The **generic search algorithm**.

The idea for the algorithm can be found here:

https://artint.info/2e/html/ArtInt2e.Ch3.S4.html

The frontier is a set of paths. Initially, the frontier contains the path of zero cost consisting of just the start node. At each step, the algorithm removes a path

if it has **found a solution** and returns the path

. Otherwise, the path is extended by one more arc by finding the neighbors and add to the frontier. This step is known as **expanding** the path

This algorithm has a few features that should be noted:

• Which path is selelected defines the search strategy. The selection of a path can affect the efficiency; see the box for more details on the use of “select”.

• It is useful to think of the *return* as a temporary return, where a caller can **retry** the search to get another answer by continuing the while loop. This can be implemented by having a class that keeps the state of the search and a search()

method that returns the next solution.

• If the procedure returns

(“**bottom**”), there are no solutions, or no remaining solutions if the search has been retried.

• The algorithm only tests if a path ends in a goal node *after* the path has been selected from the frontier, not when it is added to the frontier. There are two important reasons for this. There could be a costly arc from a node on the frontier to a goal node. The search should not always return the path with this arc, because a lower-cost solution may exist. This is crucial when the lowest-cost path is required. A second reason is that it may be expensive to determine whether a node is a goal node, and so this should be delayed in case the computation is not necessary.

If the node at the end of the selected path is not a goal node and it has no neighbours, then extending the path means removing the path from the frontier. This outcome is reasonable because this path could not be part of a path from the start node to a goal node.

```
class Node:

    def __init__(self, state, cost=0, action=None, parentNode=None, ):

        self.state = state

        self.action = action

        self.parentNode = parentNode

        self.cost = cost



    def setParentNode(self, state):

        self.parentNode = state



    def getParentNode(self):

        return self.parentNode



    def getCost(self):

        return self.cost



    def getAction(self):

        return self.action
```

```
    def getState(self):
        return self.state

    def __eq__(self, other):
        if isinstance(other, Node):
            if type(self.state) == tuple:
                return self.state[0] == other.state[0] and len(self.state[1]) == len(other.state[1])
            return self.state == other.getState()
        return False

    def __hash__(self) -> int:
        return abs(hash(self.state)) % (10 ** 8)

    def __str__(self):
        return repr(self.state)


def genericSearchAlgorithm(problem, frontier):
    expanded = []

    firstNode = Node(problem.getStartState())
    frontier.push(firstNode)
```

```
    while not frontier.isEmpty():

        current = frontier.pop()



        if problem.isGoalState(current.getState()):

            list = []

            while current.getState() != problem.getStartState():

                list.append(current.getAction())

                current = current.getParentNode()



            list.reverse()

            return list

        if current.getState() not in expanded:

            expanded.append(current.getState())

            for (nst, act, cost) in
problem.getSuccessors(current.getState()):

                node = Node(nst, current.getCost() + cost, act, current)

                frontier.push(node)
```

Question 1 - DFS - this algorithm is based on a LIFO approach because we use a stack to keep track of the nodes - as long as we still have nodes to process, we take the first item from the stack and add it to the ”visited” list - each node is stored in a tuple with its actions

It uses a queue for the generic search algorithm.

```
frontier = util.Queue()

return genericSearchAlgorithm(problem, frontier)
```

Question 2 - BFS - this algorithm is based on a FIFO approach because we use a queue to keep track of the nodes - as long as we still have nodes to process, we take the first item from the queue and add it to the ”visited” list - each node is stored in a tuple with its actions

```
frontier = util.Queue()

return genericSearchAlgorithm(problem, frontier)
```

Question 3 - UCS - this algorithm uses a priority queue to keep track of the nodes - it expands the node with the lowest cost - it searches for a goal state with the lowest cost path

```
fringe = util.PriorityQueue()

visitedList = []



fringe.push((problem.getStartState(), [], 0), 0)

(state, toDirection, toCost) = fringe.pop()

visitedList.append((state, toCost))


while not problem.isGoalState(state):
```

```
    successors = problem.getSuccessors(state)

    for son in successors:

        visitedExist = False

        total_cost = toCost + son[2]

        for (visitedState, visitedToCost) in visitedList:

            if (son[0] == visitedState) and (total_cost >= visitedToCost):

                visitedExist = True  # point recognized visited

                break



        if not visitedExist:

            fringe.push((son[0], toDirection + [son[1]], toCost + son[2]),
toCost + son[2])

            visitedList.append((son[0], toCost + son[2]))



    (state, toDirection, toCost) = fringe.pop()



return toDirection
```

Question 4 - A* - this algorithm uses a priority queue to keep track of the nodes - it is basically the UCS algorithm but we have to keep track on the heuristics given for each node - it searches for a goal state with the lowest cost path it uses. It uses a priority queue for the generic search algorithm

```
print(heuristic)

new_heuristic = lambda x: heuristic(x.getState(), problem) + x.getCost()

return genericSearchAlgorithm(problem,
util.PriorityQueueWithFunction(new_heuristic))
```

Question 7 - Eating all the dots/foodHeuristic - at first we have a function that finds the parent of a set 3 - the second function that makes the union between two sets and finds the common parent by comparing the rank of both sets - these function are used to create a path that will collect all the food - then we create a list of lists which contains the pairs of unvisited foods and the Manhattan distance between them -the next step is to perfom Kruskal algorithm for minimum spanning trees in order to find the shortest path between foods

```
foodposition = foodGrid.asList()

heuristic = [0]

for pos in foodposition:

    heuristic.append(mazeDistance(position, pos,
problem.startingGameState))

return max(heuristic)
```