Lab 4: Uninformed Search

Shi Xingyue(16337208)

November 6, 2018

Abstract

I implement uninformed search algorithms including BFS, DFS and UCS and compare them based on theoritical analysis and practical running results. To find out the searching process visually and evaluate my algorithms implement, I take use of related source codes (Pacman project) available in UC Berkeley CS188 course page.

1 Algorithms

In this experiment, I focus on these three uninformed algorithms - BFS, DFS and UCS. The first two algorithms are not cost-insensitive while the last is cost-sensitive. BFS, DFS are quite similar except they take different strategies to choose a successor to expend.

1.1 Breadth-first Search

Breadth-first Search is a simple strategy where the root node is expanded first, and then successor are expended hierarchically. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

BFS is one of the general graph-search algorithms which the shallowest unexplored node is chosen for expansion.

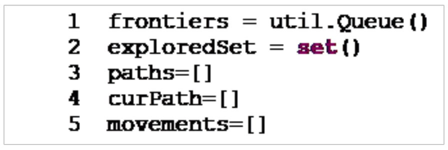
BFS can be implemented using a FIFO queue for storing frontiers, which means new nodes (which are always deeper than their parents) go to the end of queue and old nodes, which are shallower than the new nodes are popped from the head of the queue to be expanded.

The result of of BFS is the path with the least depth. Whenever the goal stated is achieved, the algorithms stops.

1.2 Depth-first Search

Depth-first Search is quite similar in principle and implementation to Breadth-first Search. However, DFS expands the deepest node in current frontier of the search tree/graph. The search first go through the deepest level of the search tree/graph, where the node is a leaf node. If this leaf node is not the goal state, it is dropped from the frontier, and the search backs up to the next deepest nodes having unexplored successors.

DFS can be implemented using a LIFO queue, or we can say, a stack. A stack means the most recently added frontier is chosen to expend,



```
frontiers = util.Queue()
exploredSet = set()
paths=[]
curPath=[]
movements=[]
```

Figure 1: Frontiers and their corresponding paths are stored separately in two containers, one Queue and the other list. This storage method makes sense here because the order of elements in two containers are easy to maintain. But things get complicated when I use Priority Queue in UCS, so I will use different data structure in UCS.

The result of of BFS is the ”leftmost” path (the solution explored first).Like BFS, DFS stops when a path to the goal state is found. It does not guarantee to find the least-cost or least-depth path.

1.3 Uniform-cost Search

BFS and DFS are cost-insensitive, while UCS is a cost-insensitive algorithm. UCS can be regarded as modified BFS. Instead of expanding the shallowest node, UCS expands the node with the lowest path cost.

UCS can be implemented by using a priority queue where the path cost is the priority.

Besides using a priority queue rather than a FIFO queue, UCS has two more modification on BFS. First, the goal test is applied when it is selected to expand rather than when it is first generated.The second is that we need another test to ensure we find the optimal solution of all.

By changing the strategy to choose the node to explore, UCS can find the optimal path. They algorithm does not stop until it tests and ensure it has found the optimal solution (even a suboptimal solution may have been found!).

2 Implementation

This project is written in python and can compiled successfully with python2.7. Library python-tk is needed to implement GUI.

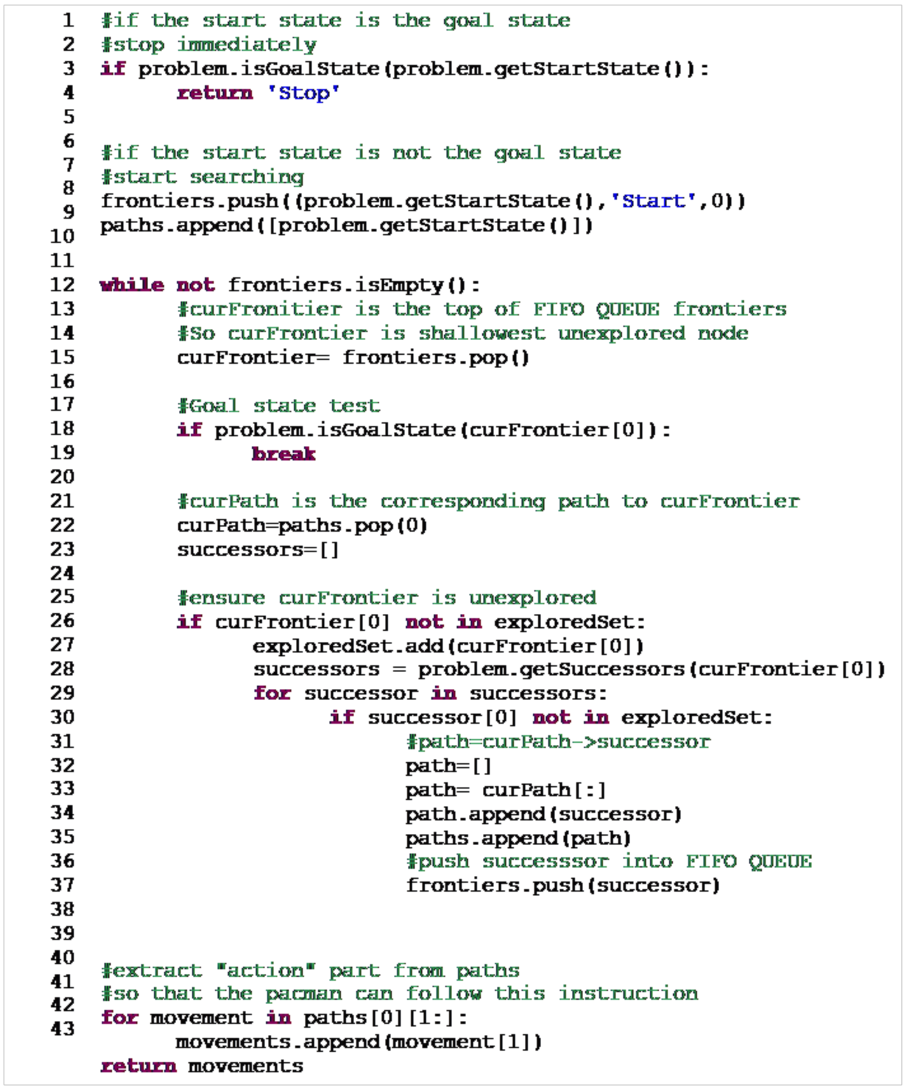
2.1 Breadth-first Search

The data structures used when implementing BFS are listed in Figure.1.

The algorithm details are listed in Figure.2. Important parts are commented in detail!

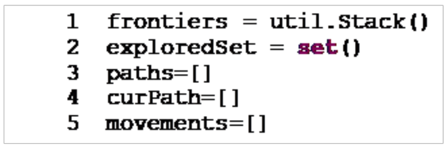
2.2 Depth-first Search

Data structures used in DFS is identical to those in BFS except for that we use a LIFO Stack, rather than a FIFO Queue to store frontiers. Details are shown in Figure.3.



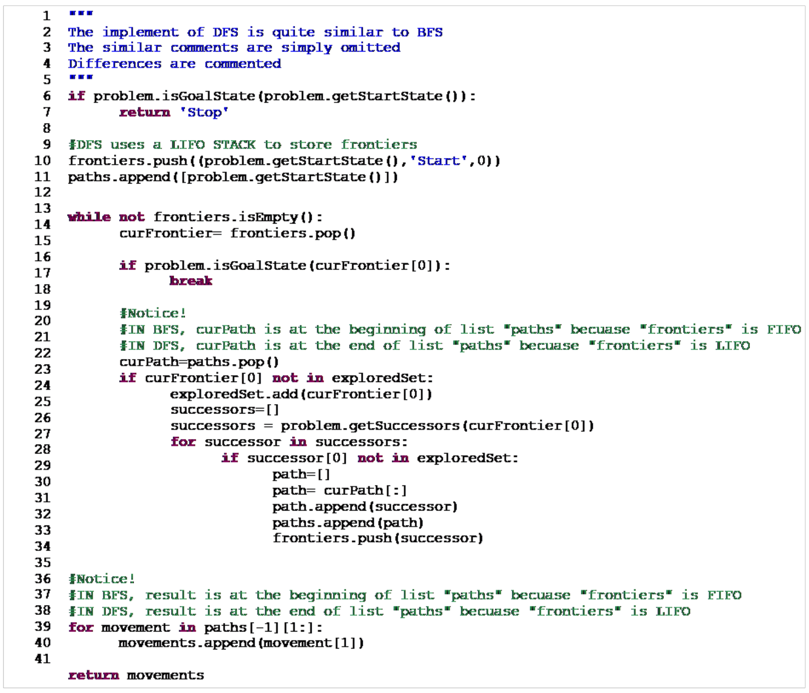
```python
#if the start state is the goal state
#stop immediately
if problem.isGoalState(problem.getStartState()):
    return 'Stop'

#if the start state is not the goal state
#start searching
frontiers.push((problem.getStartState(),'Start',0))
paths.append([problem.getStartState()])

while not frontiers.isEmpty():
    #curFronitier is the top of FIFO QUEUE frontiers
    #So curFrontier is shallowest unexplored node
    curFrontier= frontiers.pop()

    #Goal state test
    if problem.isGoalState(curFrontier[0]):
        break

    #curPath is the corresponding path to curFrontier
    curPath=paths.pop(0)
    successors=[]

    #ensure curFrontier is unexplored
    if curFrontier[0] not in exploredSet:
        exploredSet.add(curFrontier[0])
        successors = problem.getSuccessors(curFrontier[0])
        for successor in successors:
            if successor[0] not in exploredSet:
                #path=curPath->successor
                path=[]
                path= curPath[:]
                path.append(successor)
                paths.append(path)
                #push successsor into FIFO QUEUE
                frontiers.push(successor)


#extract "action" part from paths
#so that the pacman can follow this instruction
for movement in paths[0][1:]:
    movements.append(movement[1])
return movements
```

Figure 2: Implementation of BFS.



```python
frontiers = util.Stack()
exploredSet = set()
paths=[]
curPath=[]
movements=[]
```

Figure 3: In DFS, we use a LIFO Stack to store frontiers.



```
"""
The implement of DFS is quite similar to BFS
The similar comments are simply omitted
Differences are commented
"""
if problem.isGoalState(problem.getStartState()):
      return 'Stop'

#DFS uses a LIFO STACK to store frontiers
frontiers.push((problem.getStartState(),'Start',0))
paths.append([problem.getStartState()])


while not frontiers.isEmpty():
      curFrontier= frontiers.pop()

      if problem.isGoalState(curFrontier[0]):
            break

      #Notice!
      #IN BFS, curPath is at the beginning of list "paths" becuase "frontiers" is FIFO
      #IN DFS, curPath is at the end of list "paths" becuase "frontiers" is LIFO
      curPath=paths.pop()
      if curFrontier[0] not in exploredSet:
            exploredSet.add(curFrontier[0])
            successors=[]
            successors = problem.getSuccessors(curFrontier[0])
            for successor in successors:
                  if successor[0] not in exploredSet:
                        path=[]
                        path= curPath[:]
                        path.append(successor)
                        paths.append(path)
                        frontiers.push(successor)


#Notice!
#IN BFS, result is at the beginning of list "paths" becuase "frontiers" is FIFO
#IN DFS, result is at the end of list "paths" becuase "frontiers" is LIFO
for movement in paths[-1][1:]:
      movements.append(movement[1])

return movements
```

Figure 4: Implementation of DFS.

DFS's implementation is almost the same as the BFS's so I can reuse most of the BFS codes (see Figure.4). The difference is the type of container for frontiers. Besides, special attention should be paid when matching a path to its corresponding frontier, since the order of frontiers is reverse to that in BFS.

2.3 Uniform-cost Search

We can notice that the UCS algorithm is identical to BFS algorithm when all the costs are set equal. So in order to show the difference between these two algorithms. I construct another maze which is same in shape, but different in step costs to the orginal maze given by TA.

As mentioned before, it leads to chaos if I keep using the previous storage method - storing frontiers and their corresponding paths in two containers, because the the order of frontier are hard to maintain in UCS. So I take another strategy here. I store the corresponding path together with a frontier, so a corresponding path to the frontier can be easily found. The new structure of elements in "frontiers" is commented in Figure.6. Data structures used are list in Figure.5.

Searching process does not stop even when a solution is found. It will continues searching other paths and find out the optimal solution. An extra test is added at the end of the process to ensure the return value is the OPTIMAL solution (see Figure.6).

```
#frontiers has different foramt in UCS
#paths are included in frontiers to avoid chaos
frontiers = util.PriorityQueue()
exploredSet = set()
movements=[]
solutions=[]
```

Figure 5: Use a Priority Queue to store frontiers.

```
if problem.isGoalState(problem.getStartState()):
    return 'Stop'

#tuples in "frontiers" are like below:
#[(x,y), [node1,node2,...], pathcost]
frontiers.push((problem.getStartState(),[]),0)

while not frontiers.isEmpty():
    curFrontier= frontiers.pop()
    movements=curFrontier[1]
    #searching process continues even the goal state is achieved
    #extra test to ensure the path is optimal
    if problem.isGoalState(curFrontier[0]):
        solutions.append((movements, problem.getCostOfActions(movements)))
    if curFrontier[0] not in exploredSet:
        exploredSet.add(curFrontier[0])
        successors=[]
        successors = problem.getSuccessors(curFrontier[0])
        for successor in successors:
            child=successor[0]
            action=successor[1]
            path=movements[:]
            path.append(action)
            frontiers.push((child,path),problem.getCostOfActions(path))

#test whethter the path is optimal
movements=solutions[0][0]
mincost=solutions[0][1]
for solution in solutions:
    if solution[1]<mincost:
        movements=solution[0]

return movements
```

Figure 6: Implementation of UCS.

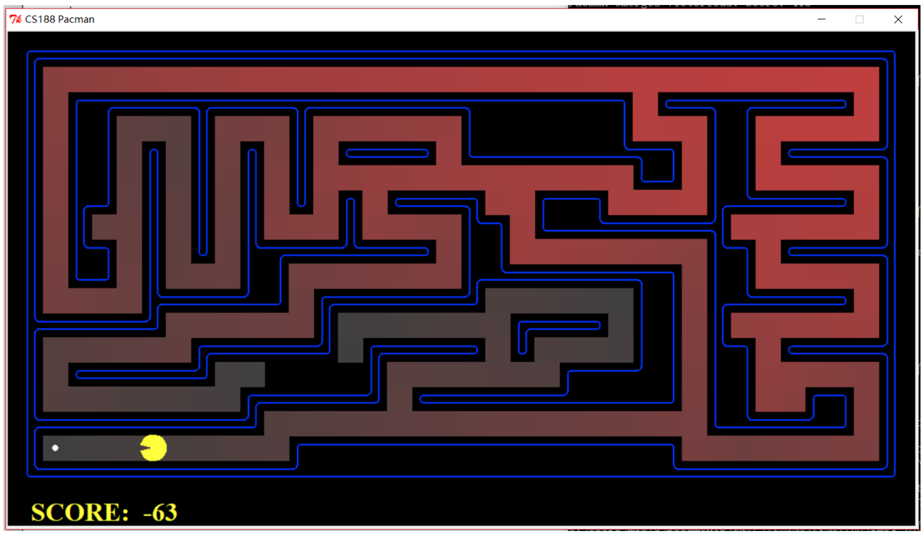


Figure 7: Running result using BFS

BFS	YES
DFS	YES if we prevent cycles
UCS	YES, it can be regarded as a modified BFS

Table 1: completeness

3 Results and Evaluation

3.1 Running Results

Running results using BFS, DFS and UCS on the unweighted maze are shown in Figure.7, Figure.8 and Figure.9. Because the cost function in this maze is always the constant value 1, the running results using BFS and UCS look same (and so are they!). We can see DFS instructs pacman to walk a much longer path than the other two algorithms, demonstrating its lacking optimality.

To test UCS on weighted maze (or else what the point of implementing it? lol) I test UCS on another two maze, DottedMaze and ScaryMaze. These two mazes have inconstant cost functions. In DottedMaze, path to a node with dot on it has lower cost than the others.In ScaryMaze, path to a node where ghost may exist has higeher cost than the others. As we can see in Figure.10 and Figure.11, pacman succeeds in choosing a ideal path! Moreover, if we use BFS or DFS to search thses two maze, pacman misses dots in DottedMaze and is caught by ghosts in ScaryMaze.

3.2 Completeness

Theoretical analysis of completeness of these three algorithms are shown in Table.1. Since I prevent cycles by maintaining "exploredSet" in DFS, the algorithm does not get trapped in a possible cycle. Three algorithms all succeed in finding a solution.

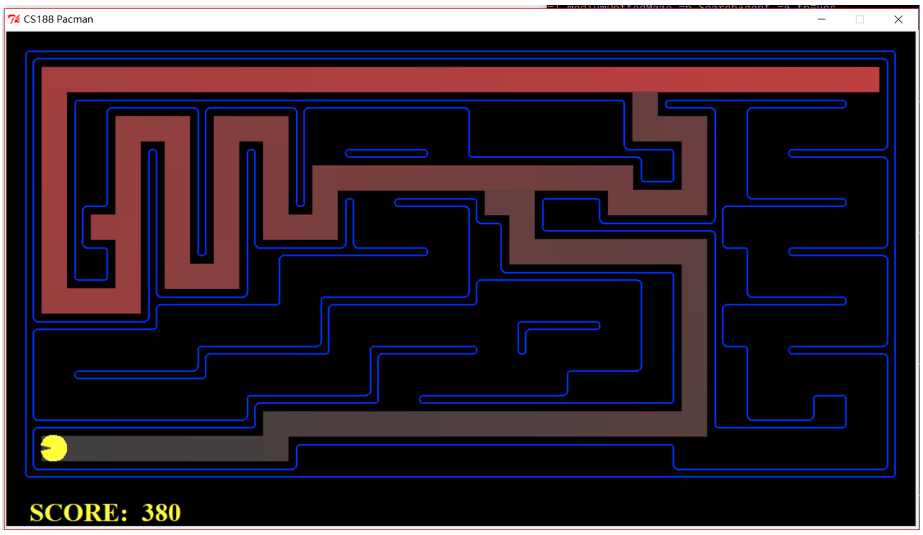


Figure 8: Running result using DFS

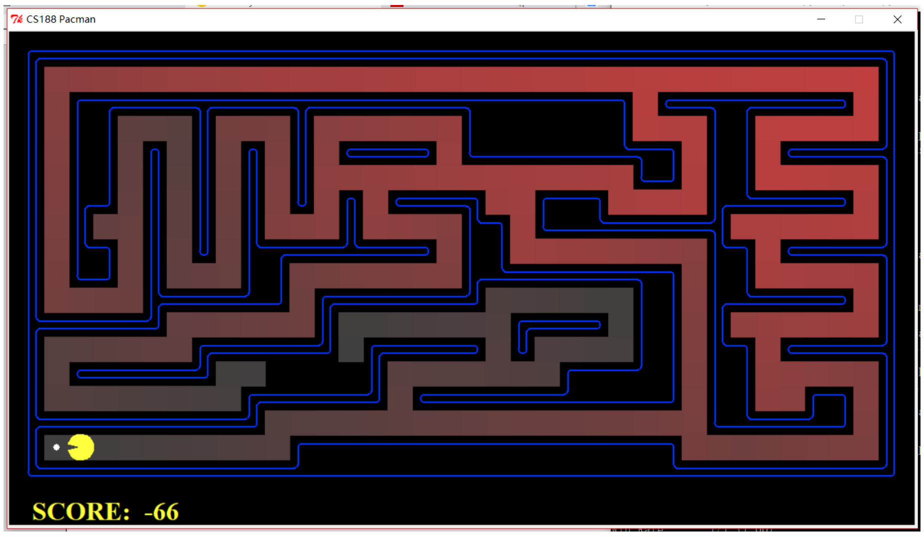


Figure 9: Running result using UCS

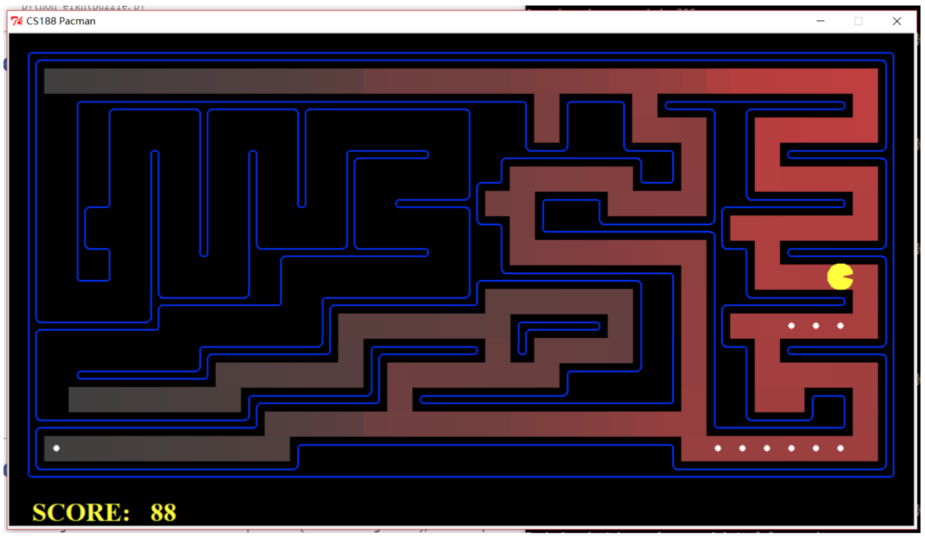


Figure 10: Running result using UCS on DottedMaze. In DottedMaze, path to a node with dot on it has lower cost than the others.

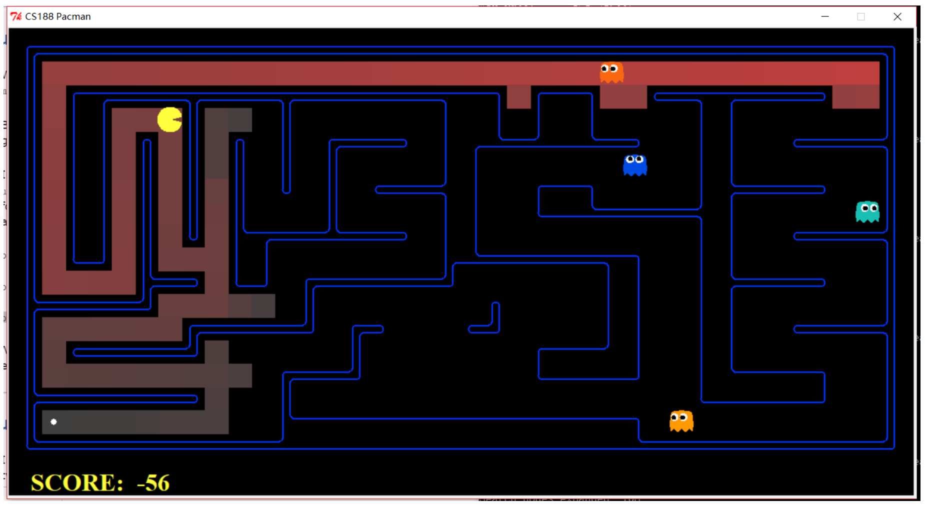


Figure 11: Running result using UCS on ScaryMaze.In ScaryMaze, path to a node where ghost may exist has higeher cost than the others.

BFS	Only if all costs are same
DFS	NO
UCS	YES

Table 2: optimality

BFS	$O(b^s)$, s is the depth of shallowest solution
DFS	$O(b^m)$, m is the height of search tree
UCS	$O(b^{C*/\epsilon})$, C* is solution cost C* and ϵ is arcs cost

Table 3: Time Complexity

3.3 Optimality

Theoretical analysis of optimality of these three algorithms are shown in Table.2. We can tell for sure now DFS do not have optimality on seeing it finds such a long path for pacman to walk in Figure.8. When all costs are same (e.g. all costs are 1), BFS can find the same solution as UCS does, namely the optimal solution (see Figure.7 and Figure.9). But when cost functions are not constant, BFS fails to find optimal solutions and thus pacman misses dots and is caught by ghost, while UCS can teaches pacman to win both games(see Figure.10 and Figure.11).

3.4 Time Complexity

Theoretical analysis of time complexity of these three algorithms are shown in Table.3. I record down the running time of these three functions (see Table.4). From the table, we can the elapsed time of BFS and DFS are very close. UCS takes more time than the other two because it introduces extra test and priority queue have more work to do when pushing and poping.

3.5 Space Complexity

Theoretical analysis of time complexity of these three algorithms are shown in Table.3. The number of explored nodes are list in Table.4. These numbers can indicates their space complexity. We can see DFS explores less nodes here, and thus store less nodes, which means it has a smaller space complexity.

4 Discussion

Based on theoretical analysis and the running results I get above, I now have deeper understanding on three algorithms' advantages, disadvantages and suitable scenarios.

BFS	0.015s
DFS	0.016s
UCS	0.031s

Table 4: Time elapsed to find a solution.

BFS	269
DFS	146
UCS	269

Table 5: The number of search nodes expanded.

BFS	$O(b^s)$, s is the depth of shallowest solution
DFS	$O(bm)$, m is the height of search tree
UCS	$O(b^{C*/\epsilon})$, C* is solution cost C* and ϵ is arcs cost

Table 6: Space Complexity

BFS is has time complexity and time complexity which are both exponential, so its scalability may not be outstanding. Besides, it is cost-insensitive. In conclusion, it is suitable for search on small scare of nodes where cost functions are constant.

DFS's striking feature is its acceptable space complexity which is much smaller than BFS and UCS. It is suitable when we need to search a giant tree and do not have strict time limit.

UCS is cost-sensitive, which means it can search on weighted search tree/graph. I assume there are much more practical scenarios to use UCS than BFS/DFS, because the cost of each action are usually different in our real life.