1. Depth First Search Implementation

   The code snippet is as follows.

   Here, we are pushing the start state into the stack and each node is checked if it's visited or not and if not then its successors are derived and their elements are pushed into the stack.

```
Depthfirst_stack = util.Stack()
first_state = problem.getStartState()
Depthfirst_stack.push(first_state)
explored = []
backtrack = {}

while Depthfirst_stack.isEmpty()!=1:
      current_state = Depthfirst_stack.pop()
      explored.append(current_state)
      if problem.isGoalState(current_state):
        reach_goal = current_state
        break
      for next_state in problem.getSuccessors(current_state):
        if next_state[0] not in explored:
          Depthfirst_stack.push(next_state[0])
          backtrack[next_state[0]] = (current_state,next_state[1])
backtrack_state = reach_goal
final = []
```

Snippet 1

```
while backtrack_state != first_state:
  final.append(backtrack[backtrack_state][1])
  backtrack_state = backtrack[backtrack_state][0]

final.reverse()
return final
```

Snippet 2

2. Breadth First Search Implementation

Here we follow more or less the same but instead of stack we use queue here.

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

    Breadthfirst_queue = util.Queue()
    first_state = problem.getStartState()
    Breadthfirst_queue.push(first_state)
    explored = []
    backtrack = {}

    while Breadthfirst_queue.isEmpty() != 1:
      current_state = Breadthfirst_queue.pop()
      explored.append(current_state)
      if problem.isGoalState(current_state):
        reach_goal = current_state
        break
```

Snippet 3

```
   for next_state in problem.getSuccessors(current_state):
     if next_state[0] not in explored:
       if next_state[0] not in backtrack:
         Breadthfirst_queue.push(next_state[0])
         backtrack[next_state[0]] = (current_state,next_state[1])

 backtrack_state = reach_goal
 final = []

 while backtrack_state != first_state:
   final.append(backtrack[backtrack_state][1])
   backtrack_state = backtrack[backtrack_state][0]

 final.reverse()
 return final()
```

Snippet 4

3. Uniform Cost Search Implementation

Here we build the path according to the cost and with the help of a Priority Queue.

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    #util.raiseNotDefined()

    first_state = problem.getStartState()
    Uniformcost_pqueue = util.PriorityQueue()
    Uniformcost_pqueue.push((first_state,1),1)
    explored = []
    backtrack = {}

    while Uniformcost_pqueue.isEmpty() != 1:
      current_state,state_wait = Uniformcost_pqueue.pop()
      explored.append(current_state)

      if problem.isGoalState(current_state):
        reach_goal = current_state
        break
```

Snippet 5

```
    for next_state in problem.getSuccessors(current_state):
      if next_state[0] not in explored:
        cost = state_wait + next_state[2]

        if next_state[0] not in backtrack:
          Uniformcost_pqueue.push((next_state[0],cost),cost)
          backtrack[next_state[0]] = (current_state,next_state[1],cost)
        elif cost<backtrack[next_state[0]][2]:
          Uniformcost_pqueue.push((next_state[0],cost),cost)
          backtrack[next_state[0]] = (current_state,next_state[1],cost)

  backtrack_state = reach_goal
  final = []

  while backtrack_state != first_state:
    final.append(backtrack[backtrack_state][1])
    backtrack_state = backtrack[backtrack_state][0]

  final.reverse()
  return final
```

Snippet 6

4. A*Search

This is similar to the UCS method but here we need to consider cost + heuristic combined and this is implemented with the help of a Priority Queue.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    #util.raiseNotDefined()

    open = util.PriorityQueue()
    first_state = problem.getStartState()
    closed_list = []
    backtrack = {}
    gscore = {}
    open.push((first_state,0),0)

    while open.isEmpty() != 1:
      current_state, state_wait = open.pop()
      if problem.isGoalState(current_state):
        reach_goal = current_state
        break

      closed_list.append(current_state)
```

Snippet 7

```
    for next_state in problem.getSuccessors(current_state):

        if next_st (variable) next_state: Any
           cost = next_state[2] + state_wait - heuristic(current_state,problem)

           if next_state[0] not in gscore or gscore[next_state[0]]>cost:

              gscore[next_state[0]] = cost
              backtrack[next_state[0]] = (current_state,next_state[1])

              gscore[next_state[0]] = cost
              fscore = cost + heuristic(next_state[0],problem)

              open.update((next_state[0],fscore),fscore)
```

Snippet 8

```
    backtrack_state = reach_goal
    final = []

    while backtrack_state != first_state:

      final.append(backtrack[backtrack_state][1])
      backtrack_state = backtrack[backtrack_state][0]

    final.reverse()
    return  final
```

Snippet 9

5. Corners Problem : Representation

```
  def getStartState(self):
      """
      Returns the start state (in your state space, not
      space)
      """
      "*** YOUR CODE HERE ***"
      #util.raiseNotDefined()
      if self.startingPosition == self.corners[0]:
        return((1,0,0,0),self.startingPosition)
      if self.startingPosition == self.corners[1]:
        return((0,1,0,0),self.startingPosition)
      if self.startingPosition == self.corners[2]:
        return((0,0,1,0),self.startingPosition)
      if self.startingPosition == self.corners[3]:
        return((0,0,0,1),self.startingPosition)

      return((0,0,0,0),self.startingPosition)
```

```
def isGoalState(self, state):
    """
    Returns whether this search state is a goal state
    """
    "*** YOUR CODE HERE ***"
    #util.raiseNotDefined()

    if state[0] == (1,1,1,1):
      return True

    return False
```

```
def getSuccessors(self, state):
    """
    Returns successor states, the actions they requi

     As noted in search.py:
        For a given state, this should return a list
        action, stepCost), where 'successor' is a su
        state, 'action' is the action required to ge
        is the incremental cost of expanding to that
    """

    successors = []
    for action in [Directions.NORTH, Directions.SOUTH
        # Add a successor state to the successor lis
        # Here's a code snippet for figuring out whe
        #   x,y = currentPosition
        #   dx, dy = Actions.directionToVector(actio
        #   nextx, nexty = int(x + dx), int(y + dy)
        #   hitsWall = self.walls[nextx][nexty]

        "*** YOUR CODE HERE ***"
        x,y = state[1]
        dx,dy = Actions.directionVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]
```

```
    hitsWall = self.walls[nextx][nexty]
    if hitsWall == False:

      state0 = state[0]
      if(nextx,nexty) == self.corners[0]:
        state0 = (1,state[0][1],state[0][2],state
      if(nextx,nexty) == self.corners[1]:
        state0 = (state[0][0],1,state[0][2],state
      if(nextx,nexty) == self.corners[2]:
        state0 = (state[0][0],state[0][1],1,state
      if(nextx,nexty) == self.corners[3]:
        state0 = (state[0][0],state[0][1],state[0

      new_state = (state0,(nextx,nexty))
      next_step = (new_state,action,1)
      successors.append(next_step)
self._expanded += 1
return successors
```