1. Uninformed Search

1.1 BFS, DFS

- The classic version of DFS (according to the course) (only parents are retained -> can revisit states multiple times, but it avoids infinite loops). This idea can also be used for BFS.
- The "optimized" versions of DFS and BFS (visited states are marked to not revisit them):

```
def BFS(init_state):
  q = Queue()
  q.push(init_state)
  viz[init_state] = 1

  while q is not empty:
    state = q.pop()

    if is_final(state):
       print(state)

    for each neigh of state:
       if is_valid(neigh) and not viz[neigh]:
          viz[neigh] = 1
          q.push(neigh)
```

```
def DFS(init_state):
  s = Stack()
  s.push(init_state)
  viz[init_state] = 1

  while s is not empty:
    state = s.pop()

    if is_final(state):
       print(state)

    for each neigh of state:
       if is_valid(neigh) and not viz[neigh]:
          viz[neigh] = 1
          s.push(neigh)
```

1.2 Uniform Cost Search (UCS)

- In BFS, nodes are visited based on the number of the transitions from the initial state
- In Uniform cost search, nodes are visited based on the distance from the initial state
- If all transitions have cost 1 => BFS = Uniform Cost Search
- Difference between Dijkstra & UCS: in Dijkstra we calculate the minimum distances between all nodes, while in UCS we calculate the minimum distances from the initial state(s) to all nodes. Uniform Cost Search is usually considered a version of Dijkstra's algorithm.

```
def uniform_cost(init_state):

   d = {}
   d [init_state] = 0
   pq = priorityQueue() #ordered by d
   pq.insert((init_state, d[init_state]))

   while pq is not empty:
        state = pq.pop() #state with the minimum d value
        pq.remove(state)
        if is_final(state): return reconstruct_path(state, came_from)

        for each neighbor of state: #transition & validation(s)functions
           if(is_valid(neighbor) and
              (neighbor not in d or d[neighbor] > d[state] + dist(neighbor, state))):

               d[neighbor] = d[state] + dist(neighbor, state)
               came_from[neighbor] = state
               pq.insert((neighbor, d[neighbor]))
   return None
```

1.3 IDDFS (Iterative Deepening Depth First Search)

- Combines the space efficiency of DFS with the fast search of states near the current state of BFS
- DFS executed in a BFS manner

```
def IDDFS(init_state, max_depth):
  for depth from 0 to max_depth:
      visited = []
      sol = depth_limited_DFS(init_state, depth, visited):
      if sol is not None:
          return sol
  return None

def depth_limited_DFS(state, depth, visited):
     if is_final(state):
        return state
     if depth == 0:
        return None
     visited.add(state)
     for each neighbor of state: #transition & validation(s)functions
         if is_valid(neighbor) and neighbor not in visited:
            res = depth_limited_DFS(neighbor, depth-1, visited)
            if res is not None:
               return res

      return None
```

1.4 BKT

- Difference from DFS: no need to retain visited states to avoid loops.
- One of the most computationally expensive strategies

```
def BKT(partial_solution):

   if (is_complete(partial_solution)): #complete = final
      return partial_solution

   for each solution in successors(partial_solution):
       if is_valid(solution):

          res = BKT(solution)
          if res:
             return res
   return None

BKT(empty_solution)
```

1.5 Bidirectional Search

- The search is done starting from both the initial state and the final state(s) with an algorithm such as BFS/DFS .
- Sometimes, it is hard to define reverse transitions to reconstruct the solution
- If BFS is used, the path determined between the initial state and a final state has minimum number of transitions

Pseudocode using BFS (Only one final state is considered. Each BFS has associated its own queue and its own visited vector):

```
def Bidirectional_search(init_state, final_state):

   f_q = Queue(); f_q.push(init_state)
   b_q = Queue(); b_q.push(final_state)
   f_viz[init_state] = 1
   b_viz[final_state] = 1
   f_came_from = {}
   b_came_from = {}

   while not f_q.empty() and not b_q.empty():

         f_state = f_q.pop()
         if(is_final(f_state) or (viz_b[f_state] == 1)):
            return reconstruct_path(f_state, f_came_from, b_came_from)

         for each neighbor of f_state: #direct transitions
            if is_valid(neighbor) and not f_viz[neighbor]:
               f_viz[neighbor] = 1
               f_q.push(neighbor)
               f_came_from[neighbor]=f_state

         b_state = b_q.pop()
         if(is_final(b_state) or (viz_f[b_state] == 1)):
            return reconstruct_path(b_state, f_came_from, b_came_from)


         for each r_neighbor of b_state: #reverse transitions
            if is_valid(r_neighbor) and not b_viz[r_neighbor]:
               b_viz[r_neighbor] = 1
               b_q.push(r_neighbor)
               b_came_from[r_neighbor]=b_state


   return None
```

2. **Informed Search**

2.1 Greedy Best First

- Evaluate **all unexplored states accessible from the current state**
- Select the unexplored state closer to the goal (the heuristic value indicates the closeness to the goal).

```
def greedy(init_state):
    pq = priorityQueue() #ordered by heuristic value
    pq.insert( (init_state, heur_val(init_state)) )
    visited = [init_state]

    while pq is not empty:
         state = pq.pop() #state with the best heuristic value
         pq.remove(state)

         if is_final(state):
             return state
         for each neighbor of state: #transition & validation(s)functions
             if is_valid(neighbor) and (neighbor not in visited):
                pq.insert( (neighbor, heur_val(neighbor)) )
                visited.add(neighbor)

    return None
```

2.2 Hill Climbing

- It is a trajectory method (at each step, only a single state is retained)
- Can get stuck in local optima
- Difference from Greedy: In HC we select the next state to be at least as good as the current one. In Greedy, we can select a next state without being better than the current one.
- Multiple ways to select of the next state from the eligible neighbors: best neighbor / first neighbor / all neighbors in order (hillclimbing-backtracking).
- There is a debate between using: h(neighbor)>= h(current_state) or h(neighbor)> h(current_state). The version used in the AI course is the first one. Because, of this, we could cycle infinitely as visited states are not marked.

```
def HC(init_state):
    state=init_state

    while(not is_final(state)):
      eligible_neighbors = []
     for each neighbor of state:
        if valid(neighbor) and h(neighbor) >= h(current_state):
           eligible_neighbors.push(neighbor)
      if eligible_neighbors is empty:
        return None
      state = choose(eligible_neighbors)
```

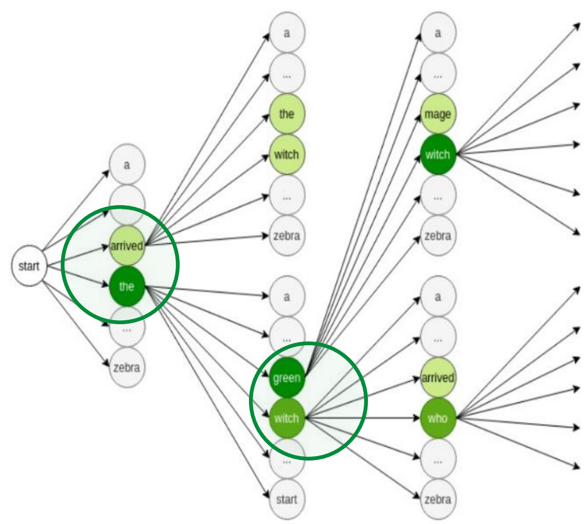
2.3 Simulated Annealing

- It is a trajectory method (at each step, only a single state is retained)
- Difference from HC: sometimes, we can go in worse states with a probability p (that decreases in time)
- Can get stuck in local optima (but is better at escaping from local optima than HC)

```
def SA(init_state):
    state=init_state
    init temperature T

    while(not stop criteria): #e.g., T>0
        neighbor = random valid neighbor of state

        if h(neighbor)>= h(current_state):
           state = neighbor

        else with probability p: #high T -> high p, low T -> low p
           state = neighbor

        update temperature T
```

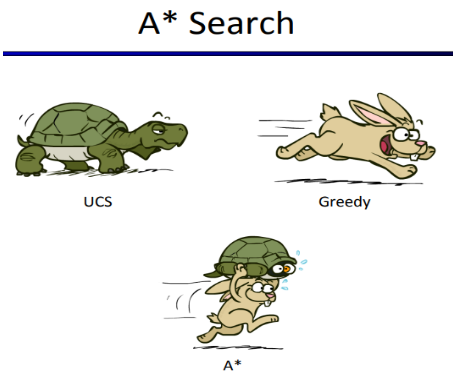
2.4 Beam Search

- Modification of BFS: only best k visited states are retained (in a *beam*), ordered based on the heuristic value
- The final state should be the first in the beam (best heuristic value)




```
def Beam_Search(init_state):
    beam = PriorityQueue()
    beam.push(init_state, h(init_state))
    viz[init_state]=1

    while(beam is not empty):

         if is_final(beam.first()):
               return beam.first()

         new_beam = []
         for state in beam:
             for neighbor of state:
                 if (is_valid(neighbor) and not viz[neighbor]):
                     viz[neighbor]=1
                     new_beam.push(neighbor, h(neighbor))

         beam = new_beam[:k]
```

2.5 A*



- Combines Uniform Cost Search + a heuristic
- At each step we consider the state S, such that the length of the path from the initial state to the goal, which passes through S, is minimum.
  - f[S] = d[S] + h(S) (the length of the path)
  - d[S] = distance from the initial state to S (updated according to Uniform Cost Search)
  - h(S) = heuristic function approximating the distance from S to the goal
- To find the shortest path from an initial state, A* needs an admissible heuristic
- "An admissible heuristic never overestimates the distance between a state and the goal."
- A consistent heuristic satisfies: h(A) <= dist(A, B) + h(B) if B is reachable from A, where:
  - h(X) = distance from state X to the goal
  - dist(X,Y) = distance between X and Y (e.g., we can consider it being the number of moves to reach Y from X).
- A consistent heuristic is also admissible.

```
def A_star(init_state):
    came_from = {}
    bestscore = -inf #folosim bestscore pt a prelua lungimea optima a drumului

    d = {}
    d[init_state] = 0
    f = {}
    f[init_state] = h(init_state)

    pq = priorityQueue() #ordered by f
    pq.insert((init_state, f[init_state]))

    while pq is not empty:
        state = pq.pop() #state with the minimum f value
        pq.remove(state)

        if is_final(state):
            if bestscore < d[state]: best_score = d[state]; best_f_state = state

        for each neighbor of state: #transition & validation(s)functions
            if is_valid(neighbor) and
              ( neighbor not in d or
                d[neighbor] > d[state] + dist(neighbor, state) ):

                d[neighbor] = d[state] + dist(neighbor, state)
                f[neighbor] = d[neighbor] + h(neighbor)
                came_from[neighbor] = state
                pq.insert((neighbor, f[neighbor]))
    return None
```

3. **Algorithms' properties**

Alg.	Always finds a solution	Solution found in min. number of transitions/ at min. distance from the initial state	Needs to mark visited states to not revisit them again	Advantages Disadvantages
Random	✗ (the algorithm is stopped after a number of steps and all solutions might be in an unexplored region)	✗	✗ (states can be revisited)	1. The path towards the solution may be very long; states may be revisited 2. Some search regions might be avoided if we stop after a certain number of transitions
optimized DFS	✓ (exception: infinite graphs)	✗	✓	1. In some cases, it can be fast even if a solution is not close to the initial state 1. Memory costly 2. May be slow even though there is a solution close to the initial state
optimized BFS	✓ (exception: infinite graphs)	✓ (minimum nb. of transitions)	✓	1. Fast if a solution is close to the initial state 1. Memory costly 2. Slow if all solutions are far away from the initial state
Uniform cost	✓ (exceptions: infinite graphs, negative cycles)	✓ (minimum distance)	✗ (states can be revisited)	1. Can determine the/a solution with minimum distance from the initial state 1. Doesn't stop if it enters a cycle with negative costs on the edges
BKT	✓ (exception: infinite graphs)	✗	✗ (it does not need to revisit states due to the way the partial solution is constructed)	1. It does not need to memorize visited states to avoid loops (revisiting states) 1. Slow approach
IDDFS	✗ (the algorithm is stopped after reaching a max. depth and all solutions might be in an unexplored region)	✓ (minimum nb. of transitions)	✓ (overall, it revisits nodes, but not in the depth limited DFS procedure)	1. Much more memory efficient than BFS. Also, the DFS is depth limited. 2. Many nodes are revisited as we increase the depth.

Bidirectional	Depends on the used version of BFS/DFS	If the used algorithm is BFS, then ✓ (minimum nb. of transitions)	✓ (two visited vectors are needed, one for each side)	1. Sometimes it is hard to define the reverse transitions 2. Needs to memorize visited states
Greedy Best First	✓ (exception: infinite graphs)	✗	✓	1. Fast strategy 2. At worst: DFS with bad choices
Hill Climbing	✗ (trajectory method)	✗	✗	1. Fastest strategy 1. Can get stuck in local optima 2. Can get stuck in infinite cycles
Simulated Annealing	✗ (trajectory method)	✗	✗	1. Fast strategy 2. Better at avoid local optima than Hill Climbing, but still can get stuck
Beam Search	✗ (only a subspace is explored)	✗	✓	1. More time and space efficient than BFS 1. Might not find a solution
A*	✓ (exception: infinite graphs, negative cycles)	✓ (minimum distance only if the heuristic is admissible)	✗ (states can be revisited)	1. Combines advantages of Greedy Best First and Uniform Cost Search 1. Might not be very time and space efficient

References

1. Uninformed and informed strategies: https://www.youtube.com/watch?v=2vPTSp7Mfhs

2. Animations (BFS, DFS, Greedy Best First, A*): https://cs.stanford.edu/people/abisee/tutorial/
https://www.redblobgames.com/pathfinding/a-star/introduction.html

https://adrianstoll.com/post/a-star-pathfinding-algorithm-animation/

3. Implementations (A*, BFS, Greedy Best First): https://www.redblobgames.com/pathfinding/a-star/implementation.html

4. Uninformed search strategies (advantages & disadvantages)

https://www.javatpoint.com/ai-uninformed-search-algorithms

5. Simulated Annealing pseudocode: http://www.cse.iitm.ac.in/~vplab/courses/optimization/SA_SEL_SLIDES.pdf

https://profs.info.uaic.ro/~eugennc/teaching/ga/