Searching Methods (Informed search)

Now that our classmates have covered uninformed search, which searches through the entire search space without knowing how far it is from the objective, we can move on to informed search, which searches through the space with additional information, such as distance from the objective, cost of moving from one state to another, etc.

For this purpose, we use a heuristic function as it helps the search by using this extra information about the search space. It takes the current state as input and determines how far it is from the objective as output. Because of this, a search algorithm that uses heuristics doesn't traverse the entire search space, finding a solution in a decent amount of time. However, for this reason, it doesn't always guarantee an optimal solution, only a good one, making it best used on very large search spaces as opposed to uninformed search, which is better used on small search spaces, as time is not as big of a constraint.

As examples, we will be exploring the hill-climbing and A* algorithms.

Hill-climbing algorithm

This algorithm has many variants. For this tutorial, we will use the *Steepest Ascent hill climbing* algorithm. It assesses every neighboring state and chooses the one that is closest to the objective, then moves on to it and iterates.

The algorithm is as follows:

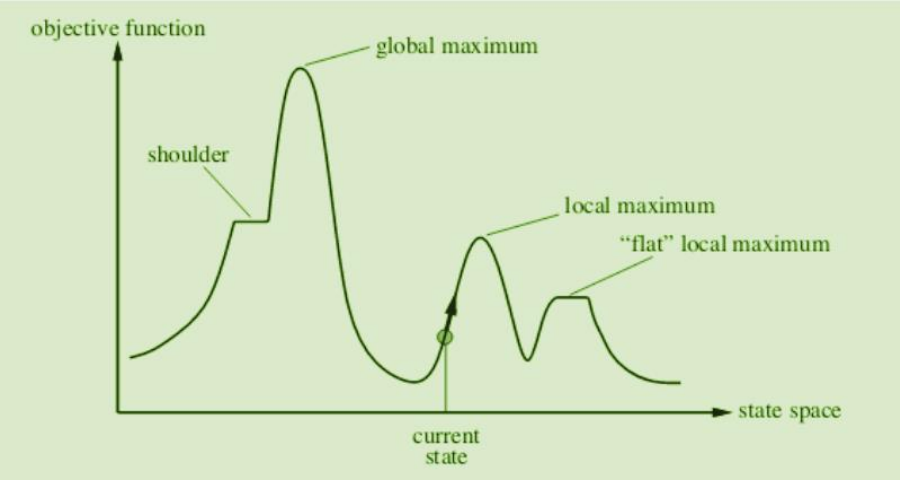
1. Evaluate initial state. If it is an objective state, stop the algorithm and return the results.
2. If not, repeat the following steps until a solution is found or there is no change in states.
    a. Evaluate the neighboring states that haven't been visited.
    b. Find the state that is closest to the objective.
    c. If the neighboring state is closer than the current state, move to it. Else, stop the algorithm and return the results.

Hill-climbing in prolog:

```
% Define the graph as a list of edges (neighbors)

edge(a, b, 1).

edge(a, c, 2).

edge(b, d, 2).

edge(b, e, 3).

edge(c, d, 4).
```

```
edge(c, f, 1).
edge(d, g, 1).
edge(e, g, 5).
edge(f, g, 3).

% Define the steepest ascent hill climbing algorithm
hill_climbing(Start, Goal, Path, Cost) :-
    hill_climbing(Start, Goal, [Start], Path, Cost).
hill_climbing(Goal, Goal, Path, Path, 0).
hill_climbing(Current, Goal, Visited, Path, Cost) :-
    findall(C-Next, (edge(Current, Next, C), not(member(Next, Visited))),
Options),
    sort(Options, SortedOptions),
    reverse(SortedOptions, DescendingOptions),
    member(C-Next, DescendingOptions),
    hill_climbing(Next, Goal, [Next|Visited], Path, Cost2),
    Cost is Cost2 + C.

% Find the path to the goal with the steepest ascent hill climbing algorithm
find_path(Start, Goal) :-
    hill_climbing(Start, Goal, Path, Cost),
    write('Path: '), write(Path),
    write(', Cost: '), write(Cost).
```

A problem for this kind of algorithm is that it tends to fall for local optimums instead of global ones. This doesn't happen on concave functions, but as a lot of functions aren't concave, it's worth considering. A diagram on GeeksforGeeks helps illustrate this:



A* algorithm

This method is one of the most popular techniques for path-finding and graph traversing. It's considered particularly smart, however, its space complexity is a major problem, as it stores all generated nodes in memory.

Assume that we are given a beginning cell and a target cell in a square grid with numerous obstacles. If at all feasible, we want to go as swiftly from the starting cell to the target cell. Here is when the A* algorithm shines.

The A* Search Algorithm selects the node at each stage based on a value, "f," which is a parameter equal to the sum of two additional factors, "g" and "h." It chooses the node or cell with the lowest "f" at each step and processes that node or cell.

Below, we define "g" and "h" as simply as possible:

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic.

The algorithm goes as follows:

1. Initialize the open list
2. Initialize the closed list
3. put the starting node on the open list (you can leave its **f** at zero)
4. while the open list is not empty
    a. find the node with the least f on the open list, call it "q"
    b. pop q off the open list
    c. generate q's 8 successors and set their parents to q
    d. for each successor

i. if successor is the goal, stop search
ii. else, compute both g and h for successor
successor.g = q.g + distance between successor and q
successor.h = distance from goal to successor
iii. if a node with the same position as successor is in the OPEN list which has a lower f than successor, skip this successor
iv. if a node with the same position as successor is in the CLOSED list which has a lower f than successor, skip this successor otherwise, add the node to the open list.

End of for loop

e. push q on the closed list

End of while loop

A* in prolog:

```
% Define the graph as a list of edges (neighbors)
edge(a, b, 1).
edge(a, c, 2).
edge(b, d, 2).
edge(b, e, 3).
edge(c, d, 4).
edge(c, f, 1).
edge(d, g, 1).
edge(e, g, 5).
edge(f, g, 3).

% Define the heuristic function that estimates the cost to reach the goal
heuristic(g, 0).
heuristic(X, H) :- edge(X, Y, C), heuristic(Y, H2), H is H2 + C.

% Define the A* search algorithm
astar(Start, Goal, Path, Cost) :-
    astar(Start, Goal, [Start], Path, Cost).

astar(Goal, Goal, Path, Path, 0).
astar(Current, Goal, Visited, Path, Cost) :-
    edge(Current, Next, C),
    not(member(Next, Visited)),
    heuristic(Next, H),
    F is C + H,
    astar(Next, Goal, [Next|Visited], Path, Cost2),
    Cost is Cost2 + C.
```

```
% Find the shortest path from start to goal
find_path(Start, Goal) :-
    astar(Start, Goal, Path, Cost),
    write('Path: '), write(Path),
    write(', Cost: '), write(Cost).
```

A major shortcoming of this algorithm is, as stated before, its space complexity, as it uses large amounts of memory. It can also be very computationally intensive if the state space is large. Moreover, if the heuristic function isn't very accurate, it can become really slow.

References

Adan, M. (n.d.). *What is the hill-climbing algorithm?* Educative: Interactive Courses for Software Developers. Retrieved February 5, 2023, from https://www.educative.io/answers/what-is-the-hill-climbing-algorithm

Belwariar, R. (2018, September 7). *A* Search Algorithm - GeeksforGeeks*. GeeksforGeeks. https://www.geeksforgeeks.org/a-search-algorithm/

Hill climbing. (2019, December 24). Wikipedia. https://en.wikipedia.org/wiki/Hill_climbing

Introduction to Hill Climbing | Artificial Intelligence. (2017, December 12). GeeksforGeeks. https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/

tutorialforbeginner.com. (n.d.). *Informed Search Algorithms | tutorialforbeginner.com*. Www.tutorialforbeginner.com. Retrieved February 1, 2023, from https://tutorialforbeginner.com/informed-search-algorithms-in-ai

What are informed search algorithms? (n.d.). Educative: Interactive Courses for Software Developers. Retrieved February 2, 2023, from https://www.educative.io/answers/what-are-informed-search-algorithms

Wikipedia Contributors. (2019, March 10). *A* search algorithm*. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/A