COMP 424 - Assignment 1

Karla Gonzalez

260715039

1 Question 1: Language Generation

We would like to design a simple language generation system which can **generate sensible and grammatically correct sentences** of English of up to length six. The system is able to generate these five words: the, cat, sat, on, mat. Thus, the sentences that we would want the system to generate are (we ignore capitalization and punctuation issues):
the cat sat
the cat on the mat sat
the cat sat on the mat
The system also incurs a cost for generating each word, equal to the number of consonants the word contains.

a) Formulate the sentence generation process as a (constructive) search problem, stating each of the parts of the search problem as shown in class. How large is the state space of this problem? Assume that sentences must contain at least one and no more than six words.

Recall that a constructive solution refers to a method which begins with an empty solution and repeatedly extends the current solution until a complete solution is obtained.

Let S : state space - all possible configurations of the domain.

We are told that the sentences must contain at least one and no more than six words and so the number of possible nodes that we have is:

$5 + 5^2 + 5^3 + 5^4 + 5^5 + 5^6 = 19530$ possible states.

This does not include our start state, which by definition of a constructive search problem is empty.

S = { "the", "cat", "sat",..."the cat sat on the mat",..., "mat mat mat mat mat mat"}

Let s_0 : initial state - the start state

We are working with a constructive solution and so our start state is empty. This will then branch out to all the possible nodes with only one word within them.

Let G : goal states - the set of end states We know that the goal states are valid, sensible and grammatically correct sentences.

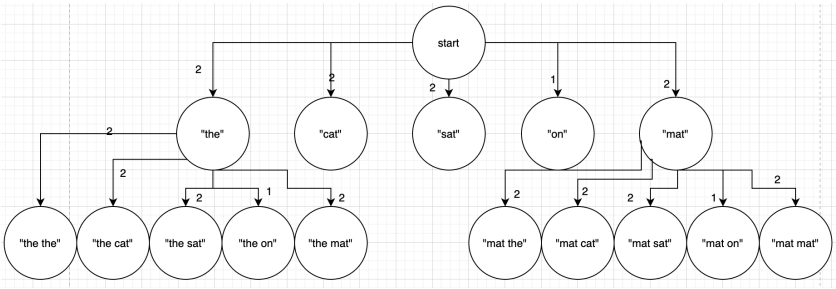
G = { "the cat sat", "the cat sat on the mat", "the cat on the mat sat"}

The goal state represents the viable, grammatically correct sentences that we want our model to generate.

Let A : operators/edges - all the actions available

A would be all the edges connecting the nodes, the actions which are taken as we traverse the graph is adding a word to the sentence (with the conditions that it is at most six words). Each edge would have a cost which corresponds to the number of consonants on the word that was added. For example, the operator from the node "the" to the node "the cat"would be the connecting edge with cost 2

b) Draw the search graph of this problem. If the graph turns out to be too large, draw a portion of it and indicate how the graph will be extended using some prose and notation. How is it different from the search tree?



This is a very brute graph but the main objective of this diagram is to show how the search graph progresses. This is just a small portion of the process. The first level after the root node have their operators denoting the cost (which is necessary when depicting a search graph) and a word of the 5 possibilities is added, this is how we obtain the first level of the graph. Similarly the second level will have all 2 word 'sentences' starting with the above prefixes. This diagram only shows the expansion of the outer two nodes. Not shows are the remaining 4 levels (for a total of 6 levels, not including the root). The third level will be all 3 word 'sentences' starting with the prefixes determined by their parent.

Recall that a search tree corresponds to the explorations of a graph where each node has a state id and the cost of the path so far. It may also include additional information such as the parent state, the operator used to generate it and even the depth of the node (slide 17, lecture 2). Whereas the search graph (map) corresponds to the possible states in which you may be in. For this problem, the diagrams would look very similar with only the differences stated above making them distinguishable

c) Trace the run of the search process using the following algorithms for up to 10 steps of the algorithm. Given multiple states to explore that are otherwise equivalent in priority, the algorithm should prefer to generate the word that comes first alphabetically.

1. Breadth-First Search (BFS) Recall that BFS follows a FIFO approach. We use a queue where we enqueue nodes at the end of the queue. All nodes at level i get expanded before all nodes at level i+1.

Start: {(S,0)}
Iteration 1:
~~{(S,0)}~~,(cat,2), (mat,2), (on,1), (sat,2), (the,2)}
Iteration 2:
~~{(S,0)},(cat,2)~~, (mat,2), (on,1), (sat,2), (the,2), (cat cat,4), (cat mat 4), (cat on, 3), (cat sat,4), (cat the,4)}
Iteration 3:
{~~(S,0),(cat,2), (mat,2)~~, (on,1), (sat,2), (the,2), (cat cat,4), (cat mat,4), (cat on, 3), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat on, 3), (mat sat,4), (mat the,4)}
Iteration 4:
~~{(S,0),(cat,2), (mat,2), (on,1),~~ (sat,2), (the,2), (cat cat,4), (cat mat,4), (cat on, 3), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat on, 3), (mat sat,4), (mat the,4), (on cat, 3), (on mat, 3), (on on, 2), (on sat, 3), (on the, 3)}
Iteration 5:
~~{(S,0),(cat,2), (mat,2), (on,1), (sat,2),~~ (the,2), (cat cat,4), (cat mat,4), (cat on, 3), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat on, 3), (mat sat,4), (mat the,4), (on cat, 3), (on mat, 3), (on on, 2), (on sat, 3), (on the, 3), (sat cat,4), (sat mat,4), (sat on, 3), (sat sat,4), (sat the,4) }
Iteration 6:
~~{(S,0),(cat,2), (mat,2), (on,1), (sat,2), (the,2),~~ (cat cat,4), (cat mat,4), (cat on, 3), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat on, 3), (mat sat,4), (mat the,4), (on cat, 3), (on mat, 3), (on on, 2), (on sat, 3), (on the, 3), (sat cat,4), (sat mat,4), (sat on, 3), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the on, 3), (the sat,4), (the the,4)}
Iteration 7:
~~{(S,0),(cat,2), (mat,2), (on,1), (sat,2), (the,2), (cat cat,4)~~, (cat mat,4), (cat on, 3), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat on, 3), (mat sat,4), (mat the,4), (on cat, 3), (on mat, 3), (on on, 2), (on sat, 3), (on the, 3), (sat cat,4), (sat mat,4), (sat on, 3), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the on, 3), (the sat,4), (the the,4), (cat cat cat,6), (cat cat mat,6), (cat cat on,5), (cat cat sat,6), (cat cat the,6)}
Iteration 8:
~~{(S,0),(cat,2), (mat,2), (on,1), (sat,2), (the,2), (cat cat,4), (cat mat,4)~~, (cat on, 3), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat on, 3), (mat sat,4), (mat the,4), (on cat, 3), (on mat, 3), (on on, 2), (on sat, 3), (on the, 3), (sat cat,4), (sat mat,4), (sat on, 3), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the on, 3), (the sat,4), (the the,4), (cat cat cat,6), (cat cat mat,6), (cat cat on,5), (cat cat sat,6), (cat cat the,6), (cat mat cat,6), (cat mat mat,6), (cat mat on,5), (cat mat sat,6), (cat mat the,6)}

Iteration 9:
~~{(S,0),(cat,2), (mat,2), (on,1),~~ ~~(sat,2), (the,2), (cat cat,4),~~ ~~(cat mat,4)~~, ~~(cat on, 3),~~ (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat on, 3), (mat sat,4), (mat the,4), (on cat, 3), (on mat, 3), (on on, 2), (on sat, 3), (on the, 3), (sat cat,4), (sat mat,4), (sat on, 3), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the on, 3), (the sat,4), (the the,4), (cat cat cat,6), (cat cat mat,6), (cat cat on,5), (cat cat sat,6), (cat cat the,6), (cat mat cat,6), (cat mat mat,6), (cat mat on,5), (cat mat sat,6), (cat mat the,6), (cat on cat,5), (cat on mat,5), (cat on on,4), (cat on sat,5), (cat on the,5)}
Iteration 10:
~~{(S,0),(cat,2), (mat,2), (on,1),~~ ~~(sat,2), (the,2), (cat cat,4),~~ ~~(cat mat,4)~~, ~~(cat on, 3),~~ ~~(cat sat,4)~~, (cat the,4), (mat cat, 4),(mat mat 4), (mat on, 3), (mat sat,4), (mat the,4), (on cat, 3), (on mat, 3), (on on, 2), (on sat, 3), (on the, 3), (sat cat,4), (sat mat,4), (sat on, 3), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the on, 3), (the sat,4), (the the,4), (cat cat cat,6), (cat cat mat,6), (cat cat on,5), (cat cat sat,6), (cat cat the,6), (cat mat cat,6), (cat mat mat,6), (cat mat on,5), (cat mat sat,6), (cat mat the,6), (cat on cat,5), (cat on mat,5), (cat on on,4), (cat on sat,5), (cat on the,5), (cat sat cat,6), (cat sat mat,6), (cat sat on,5), (cat sat sat,6), (cat sat the,6)}

2. Uniform Cost Search Recall that Uniform Cost Search is basically BFS fixed to ensure an optimal path with general step cost. We will use a priority queue instead of a simple queue where we will insert nodes in the increasing order of the cost of the paths so far. Recall that general cost = where actions may have different cost.

Start: {(S,0)}
Iteration 1:
{~~(S,0)~~, (on,1), (cat,2), (mat,2), (sat,2), (the,2)}
Iteration 2:
{~~(S,0), (on,1)~~, (cat,2), (mat,2), (sat,2), (the,2), (on on, 2), (on cat, 3), (on mat, 3), (on sat, 3), (on the, 3)}
Iteration 3:
{~~(S,0), (on,1), (cat,2)~~, (mat,2), (sat,2), (the,2), (on on, 2), (on cat, 3), (on mat, 3), (on sat, 3), (on the, 3), (cat on, 3), (cat cat,4), (cat mat,4), (cat sat,4), (cat the,4),}
Iteration 4:
{~~(S,0), (on,1), (cat,2), (mat,2)~~, (sat,2), (the,2), (on on, 2), (on cat, 3), (on mat, 3), (on sat, 3), (on the, 3), (cat on, 3), (mat on, 3), (cat cat,4), (cat mat,4), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat sat,4), (mat the,4)}
Iteration 5:
{~~(S,0), (on,1), (cat,2), (mat,2), (sat,2)~~, (the,2), (on on, 2), (on cat, 3), (on mat, 3), (on sat, 3), (on the, 3), (cat on, 3), (mat on, 3), (sat on, 3), (cat cat,4), (cat mat,4), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat sat,4), (mat the,4),(sat cat,4), (sat mat,4), (sat sat,4), (sat the,4)}
Iteration 6:
{~~(S,0), (on,1), (cat,2), (mat,2), (sat,2), (the,2),~~ (on on, 2), (on cat, 3), (on mat, 3), (on sat, 3), (on the, 3), (cat on, 3), (mat on, 3), (sat on, 3), (the on, 3), (cat cat,4), (cat mat,4), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat sat,4), (mat the,4),(sat cat,4), (sat mat,4), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the sat,4), (the the,4)}
Iteration 7:
{~~(S,0), (on,1), (cat,2), (mat,2), (sat,2), (the,2), (on on, 2)~~, (on cat, 3), (on mat, 3), (on sat, 3), (on the, 3), (cat on, 3), (mat on, 3), (sat on, 3), (the on, 3), (on on on,3), (cat cat,4), (cat mat,4), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat sat,4), (mat the,4),(sat cat,4), (sat mat,4), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the sat,4), (the the,4), (on on cat,4), (on on mat,4),(on on sat,4), (on on the,4)}
Iteration 8:
{~~(S,0), (on,1), (cat,2), (mat,2), (sat,2), (the,2), (on on, 2), (on cat, 3)~~, (on mat, 3), (on sat, 3), (on the, 3), (cat on, 3), (mat on, 3), (sat on, 3), (the on, 3), (on on on,3), (cat cat,4), (cat mat,4), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat sat,4), (mat the,4),(sat cat,4), (sat mat,4), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the sat,4), (the the,4), (on on cat,4), (on on mat,4),(on on sat,4), (on on the,4), (on cat on, 4), (on cat cat, 5), (on cat mat, 5), (on cat sat, 5), (on cat the, 5)}
Iteration 9:
{~~(S,0), (on,1), (cat,2), (mat,2), (sat,2), (the,2), (on on, 2), (on cat, 3), (on mat, 3)~~, (on sat, 3), (on the, 3), (cat on, 3), (mat on, 3), (sat on, 3), (the on, 3), (on on on,3), (cat cat,4), (cat mat,4), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat sat,4), (mat the,4),(sat cat,4), (sat mat,4), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the sat,4), (the the,4), (on on cat,4), (on on mat,4),(on on sat,4), (on on the,4), (on cat on, 4), (on mat on, 4), (on cat cat, 5), (on cat mat, 5), (on cat sat, 5), (on cat the, 5), (on mat cat, 5), (on mat mat, 5), (on mat sat, 5), (on mat the, 5)}
Iteration 10:
{~~(S,0), (on,1), (cat,2), (mat,2), (sat,2), (the,2), (on on, 2), (on cat, 3), (on mat, 3), (on sat, 3),~~ (on the, 3), (cat on, 3), (mat on, 3), (sat on, 3), (the on, 3), (on on on,3), (cat cat,4), (cat mat,4), (cat sat,4), (cat the,4), (mat cat, 4),(mat mat 4), (mat sat,4), (mat the,4),(sat cat,4), (sat mat,4), (sat sat,4), (sat the,4), (the cat,4), (the mat,4), (the sat,4), (the the,4), (on on cat,4), (on on mat,4),(on on sat,4), (on on the,4), (on cat on, 4), (on mat on, 4), (on sat on,4), (on cat cat, 5), (on cat mat, 5), (on cat sat, 5), (on cat the, 5), (on mat cat, 5), (on mat mat, 5), (on mat sat, 5), (on mat the, 5), (on sat cat,6), (on sat mat,6), (on sat sat,6), (on sat the,6)}

3. Depth First Search (DFS) DFS uses a LIFO approach, meaning that we use a stack instead of a queue like BFS. Nodes at the deepest level are expanded before shallower ones.

Start: {(S,0)}
Iteration 1:
{~~(S,0)~~, (cat,2), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 2:
{~~(S,0)~~, ~~(cat,2)~~, (cat cat,4), (cat mat,4), (cat on,3), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 3:
{~~(S,0)~~, ~~(cat,2)~~, ~~(cat cat,4)~~, (cat cat cat,6), (cat cat on,5), (cat cat mat,6), (cat cat sat,6), (cat cat the,6), (cat mat,4), (cat on,3), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 4:
{~~(S,0)~~, ~~(cat,2)~~, ~~(cat cat,4)~~, ~~(cat cat cat,6)~~, (cat cat cat cat,8), (cat cat cat on,7), (cat cat cat mat,8), (cat cat cat sat,8), (cat cat cat the,8), (cat cat on,5), (cat cat mat,6), (cat cat sat,6), (cat cat the,6), (cat mat,4), (cat on,3), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 5:
{~~(S,0)~~, ~~(cat,2)~~, ~~(cat cat,4)~~, ~~(cat cat cat,6)~~, ~~(cat cat cat cat,8)~~, (cat cat cat cat cat,10), (cat cat cat cat on,9), (cat cat cat cat mat,10), (cat cat cat cat sat,10), (cat cat cat cat the,10), (cat cat cat on,7), (cat cat cat mat,8), (cat cat cat sat,8), (cat cat cat the,8), (cat cat on,5), (cat cat mat,6), (cat cat sat,6), (cat cat the,6), (cat mat,4), (cat on,3), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 6:
{~~(S,0)~~, ~~(cat,2)~~, ~~(cat cat,4)~~, ~~(cat cat cat,6)~~, ~~(cat cat cat cat,8)~~, ~~(cat cat cat cat cat,10),~~ (cat cat cat cat cat,12), (cat cat cat cat cat on,11), (cat cat cat cat cat mat,12), (cat cat cat cat cat sat,12), (cat cat cat cat cat the,12), (cat cat cat cat on,9), (cat cat cat cat mat,10), (cat cat cat cat sat,10), (cat cat cat cat the,10), (cat cat cat on,7), (cat cat cat mat,8), (cat cat cat sat,8), (cat cat cat the,8), (cat cat on,5), (cat cat mat,6), (cat cat sat,6), (cat cat the,6), (cat mat,4), (cat on,3), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 7:
{~~(S,0)~~, ~~(cat,2)~~, ~~(cat cat,4)~~, ~~(cat cat cat,6)~~, ~~(cat cat cat cat,8)~~, ~~(cat cat cat cat cat,10)~~, ~~(cat cat cat cat cat cat,12),~~ (cat cat cat cat cat on,11), (cat cat cat cat cat mat,12), (cat cat cat cat cat sat,12), (cat cat cat cat cat the,12), (cat cat cat cat on,9), (cat cat cat cat mat,10), (cat cat cat cat sat,10), (cat cat cat cat the,10), (cat cat cat on,7), (cat cat cat mat,8), (cat cat cat sat,8), (cat cat cat the,8), (cat cat on,5), (cat cat mat,6), (cat cat sat,6), (cat cat the,6), (cat mat,4), (cat on,3), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 8:
{~~(S,0)~~, ~~(cat,2)~~, ~~(cat cat,4)~~, ~~(cat cat cat,6)~~, ~~(cat cat cat cat,8)~~, ~~(cat cat cat cat cat,10)~~, ~~(cat cat cat cat cat cat,12)~~, ~~(cat cat cat cat cat on,11)~~, (cat cat cat cat cat mat,12), (cat cat cat cat cat sat,12), (cat cat cat cat cat the,12), (cat cat cat cat on,9), (cat cat cat cat mat,10), (cat cat cat cat sat,10), (cat cat cat cat the,10), (cat cat cat on,7), (cat cat cat mat,8), (cat cat cat sat,8), (cat cat cat the,8), (cat cat on,5), (cat cat mat,6), (cat cat sat,6), (cat cat the,6), (cat mat,4), (cat on,3), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 9:
{~~(S,0)~~, ~~(cat,2)~~, ~~(cat cat,4)~~, ~~(cat cat cat,6)~~, ~~(cat cat cat cat,8)~~, ~~(cat cat cat cat cat,10)~~, ~~(cat cat cat cat cat cat,12)~~, ~~(cat cat cat cat cat on,11)~~, ~~(cat cat cat cat cat mat,12),~~ (cat cat cat cat cat sat,12), (cat cat cat cat cat the,12), (cat cat cat cat on,9), (cat cat cat cat mat,10), (cat cat cat cat sat,10), (cat cat cat cat the,10), (cat cat cat on,7), (cat cat cat mat,8), (cat cat cat sat,8), (cat cat cat the,8), (cat cat on,5), (cat cat mat,6), (cat cat sat,6), (cat cat the,6), (cat mat,4), (cat on,3), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 10:
{~~(S,0)~~, ~~(cat,2)~~, ~~(cat cat,4)~~, ~~(cat cat cat,6)~~, ~~(cat cat cat cat,8)~~, ~~(cat cat cat cat cat,10)~~, ~~(cat cat cat cat cat cat,12)~~, ~~(cat cat cat cat cat on,11)~~, ~~(cat cat cat cat cat mat,12)~~, ~~(cat cat cat cat cat sat,12),~~ (cat cat cat cat cat the,12), (cat cat cat cat on,9), (cat cat cat cat mat,10), (cat cat cat cat sat,10), (cat cat cat cat the,10), (cat cat cat on,7), (cat cat cat mat,8), (cat cat cat sat,8), (cat cat cat the,8), (cat cat on,5), (cat cat mat,6), (cat cat sat,6), (cat cat the,6), (cat mat,4), (cat on,3), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}

It is important to note that the last few iterations i interpreted as popping a node if it had no children, i could have kept going, following the pattern that is established for DFS which is expanding the nodes which you pop off the stack.

4. Iterative Deepening Search (IDS) We do depth limited search (specify a maximum depth and terminate if goal state is found of max depth is

reached) but with increasing depth. This method is complete and has linear memory requirements.

Limit = 0
{(S,0)}
Limit = 1
Iteration 1:
{(S,0)}
Iteration 2:
~~{(S,0)~~, (cat,2), (on,1), (mat,2), (sat,2), (the,2)}
Limit = 2
Iteration 3:
{(S,0)}
Iteration 4:
~~{(S,0)~~, (cat,2), (on,1), (mat,2), (sat,2), (the,2)}
Iteration5:
~~{(S,0)~~, ~~(cat,2)~~, (cat cat,4), (cat on,3), (cat mat,4), (cat sat,4), (cat the,4), (on,1), (mat,2), (sat,2), (the,2)}
Iteration 6:
~~{(S,0)~~, ~~(cat,2)~~, (cat cat,4), (cat on,3), (cat mat,4), (cat sat,4), (cat the,4), ~~(on,1),~~ (on cat,3), (on on,2), (on mat,3), (on sat,3), (on the,3), (mat,2), (sat,2), (the,2)}
Iteration 7:
~~{(S,0), (cat,2),(on,1)~~ ~~(mat,2)~~, (mat cat,4), (mat on,3), (mat mat,4), (mat sat,4), (mat the,4) (sat,2), (the,2)}
Iteration 8:
~~{(S,0)~~, ~~(cat,2)~~, ~~(mat,2)~~, ~~(sat,2)~~, (sat cat,4), (sat on,3), (sat mat,4), (sat sat,4), (sat the,4), (the,2)}
Iteration 9:
~~{(S,0)~~, ~~(cat,2)~~, ~~(mat,2)~~, ~~(sat,2), (the,2)~~, (the cat,4), (the on,3), (the mat,4), (the sat,4), (the the,4),
Limit =3
Iteration 10:
{(S,0)}

It is important to note that IDS expands nodes multiple times (as the depth limit increases) but the computation time has same complexity.

d) Describe a scheme that uses local search for this problem, such that given enough running time, the algorithm would find a goal state. Be sure to state each of the parts of the local search algorithm as shown in class.

We could implement hill climbing which would always reach a goal state, given enough time. We know that there are 3 possible goal states, all which have the same evaluation function value, meaning that in the context of hill climbing, all have their maxima with the same vales. This means that there are 3 global maxima. The set of states are all sentences in the state space which was stated above. We are given three goal states and the search graph. The goal is to arrive at one of the three goal states. We can define the evaluation function as the number of words that align with any of the 3 goal states in both word and position divided by the total number of words in that goal state (goal state = GS).

$$E(X) = max\{\frac{wordsmatchGS1}{totalwordsGS1}, \frac{wordsmatchGS2}{totalwordsGS2}, \frac{wordsmatchGS2}{totalwordsGS2}\} \quad (1)$$

The algorithm starts from a random configuration. At each step E(x) is evaluated for each neighbour. If a neighbours evaluation function is higher than the nodes value, the neighbour becomes the next starting point in the next step. A goal state is obtained when E(X)=1.

2 Question 2: Search Algorithms

a) Describe a state space in which iterative deepening search performs much worse than depth-first search ie. $O(n^2)$ **vs** $O(n)$

We know that Iterative Deepening Search:

- We do depth limited search (specify a maximum depth and terminate if goal state is found of max depth is reached) but with increasing depth.
- This algorithm will expand nodes multiple times as the limit is increased.
- Has linear memory requirements like DFS. Linear space complexity refers to $O(bm)$
  - b: branching factor is the maximum number of operators which can be applied at any time.
- Optimal and Complete: Will find optimal solution for problems with unit step costs (equal weights on edges). this will always find a solution because it traverses each level as the limits increase.

We know that Depth First Search:

- Implemented with a stack - nodes at the deepest level are expanded before the shallower ones.
- Worst case has exponential time complexity (as do all other uninformed search algorithms).
- Not optimal, may find a more costly solution before a solution at a shallower node due to the way it traverses the tree. This also leads to the issue that it may not be complete depending on the depth of the tree.

Let us consider the case where we have a graph where each node, including the root only has one child. Assume there are n nodes. Let the goal states be the leaf of the graph. Then DFS would proceed linearly along each child until it reached the leaf. This would take $O(n)$ time. With IDS, in the first iteration you will only visit the child of the root. In the second iteration, you will visit the root's child and its own child (depth = 2, visited 2 nodes). In the third iteration, you go to a depth of 3, visiting 3 nodes. Hence the total number of visits is 1 + 2 + ... + n = $O(n^2)$.

This argument can also be applied to a problem with a large branching factor, a reasonable depth and where the goal states are the leaves. DFS would proceed to each level with each iteration, reaching the leaf in $O(d)$. IDS would have to visit all the branches at each level ontop of having to revisit each level each time the limit is increased. This would perform much worse than DFS.

Prove each of the following or give counter examples

b) Breadth first search is a special case of Uniform Cost Search

We know that Uniform Cost Search is an algorithm which uses BFS with a priority queue instead, inserting nodes in the increasing order of the cost of the path so far. This works best when we have general step costs (different costs for different actions). Thus if we have unit cost steps, we get BFS.

c) Uniform-cost search is a special case of $A*$ search.

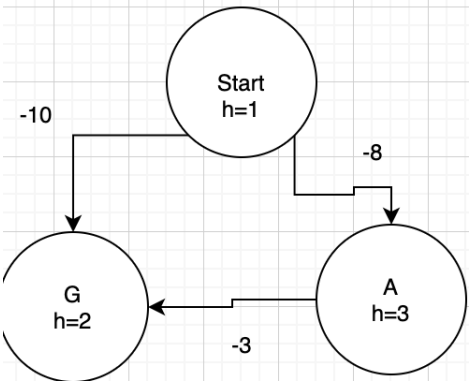
$A*$ Search is a heuristic search with an admissible heuristic.

- Heuristic: Cost to go
- Admissible Heuristic: $h(n) =¡ h*(n)$ for all n. Meaning that your heuristic estimate $h(n)$ is smaller or equal to the shortest path.

Recall that Uniform Cost Search uses a priority queue to ensure an optimal path with general step costs. This means that we perform the search with respect to $f(n) = g(n)$, the evaluation function of the overall cost. Comparing this to $A*$ Search and its overall function $f(n) = g(n) + h(n)$. Then it is clear that UCS is the case of $A*$ when the heuristic function is zero. This is a valid heuristic because it is admissible because it always "underestimates" the distance to the goal, which cannot be smaller than 0.

d) $A*$ search is optimal in the case where negative edge weights are allowed.

This is false. Consider the following example



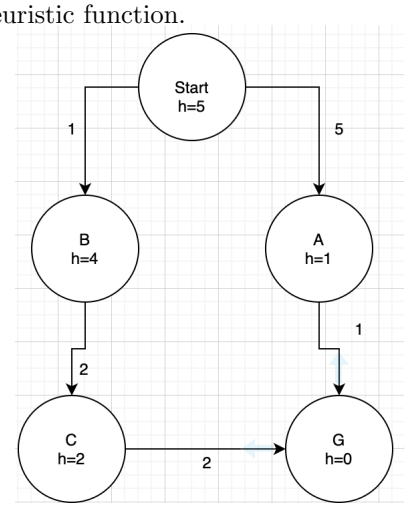
Following the heuristic search you would first find G with a weight of -10. Then as you traversed your queue you would find the optimal solution to be a cost of -11, because it is smaller than the original path cost that you found.

e) Best-first search is optimal in the case where we have a perfect heuristic (i.e. $h(n) = h*(n)$ the true cost to the closest goal state)

We know that Best-first Search is an algorithm which expands the most promising node according to the heuristic. With this definition alone BestFS is not

optimal. We know that the better the heuristic, the more efficient it is. If the heuristic is always 0, then Bestfirst Search is the same as BreadthFS, which is optimal due to the way it traverses a graph and the definition of an optimal solution.

But we can also come up with a counter example for when we have a perfect heuristic function.



In the above example, doing BestFS would lead you on the following path to the solution {(Start,5), (A,1),(B,1), (G,6)}. But if you were to continue, you would see that you would arrive at a more efficient solution as (G,3). Thus this algorithm is not optimal for a perfect heuristic.

f) Suppose there is a unique optimal solution. Then, $A*$ search with a perfect heuristic will never expand nodes that are not in the path of the optimal solution.

If the heuristic function is not admissible, than we can have an estimation that is bigger than the actual path cost from some node to a goal node. If this higher path cost estimation is on the least cost path (that we are searching for), the algorithm will not explore it and it may find another (not least cost) path to the goal. Also by the proof on slide 31 of Lecture 3 we show that all nodes on the optimal path will be chosen before any other node and thus any other, non-optimal solution.

3 Question 3: Optimization

You should write some code to solve this question. Consider the following functions:

$$f_1(x, y) = sin(2x) + cos(y/2) \tag{2}$$

$$f_2(x, y) = |x - 2| + |0.5 * y + 1| - 4 \tag{3}$$

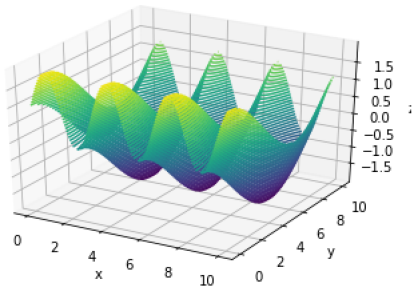
We would like to maximize these functions within the range of $0 =< x, y =< 10$. For each part below and each setting, report the mean and standard deviation of the number of steps to convergence and of the final value f_1^* and f_2^* each case. Use plots and/or tables to report your results in an organized manner.

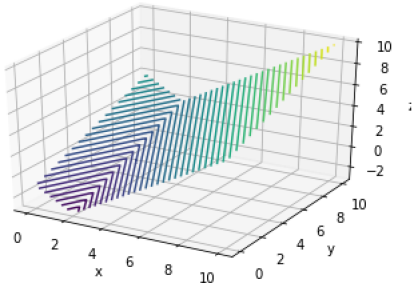
a) For each function, apply hill climbing, starting from 100 random points in the range. Repeat this procedure for each choice of step size in [0.01, 0.05, 0.1, 0.2]. A neighbour is a point where x and/or y has increased or decreased by the stepsize; i.e., there are up to 8 neighbours from any given point. What patterns do you see?

```
#!/usr/bin/env python
# coding: utf-8

import numpy as np
import pandas as pd
import random as rand
import matplotlib.pyplot as plt
from statistics import stdev
from statistics import mean
from operator import itemgetter


#3a)

def f1(x, y):
    return (np.sin(2*x) + np.cos(y/2))

def f2(x, y):
    return (np.absolute(x-2) + np.absolute(0.5*y+1) - 4)

#Plot f1

x = np.linspace(0, 10, 100)
y = np.linspace(0, 10, 100)

X, Y = np.meshgrid(x, y)
Z = f1(X, Y)
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');

#plot f2

x = np.linspace(0, 10, 100)
y = np.linspace(0, 10, 100)

X, Y = np.meshgrid(x, y)
Z = f2(X, Y)
fig = plt.figure()
ax = plt.axes(projection='3d')
```

```
ax.contour3D(X, Y, Z, 50)
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');

##Select the best successor for a point (x, y) under a function f()

def best_neighbor(s,f,x,y):
    x_ = max(x-s,0)
    y_ = max(y-s,0)
    xp = min(x+s,10)
    yp = min(y+s,10)
    num = [x_,y,f(x_,y)]
    l = [[x_,y, f(x_,y)],[x_,y_, f(x_,y_)],[x_,yp, f(x_,yp)],[x,y_,
     f(x,y_)],[x,yp, f(x,yp)],[xp,yp, f(xp,yp)],[xp,y_, f(xp,y_)]]
    for x in l:
        if x[2]>num[2]:
            num = x
    return num[0], num[1];

#Main algorithm

def hill_climb(s,f,x0,y0):
    x = x0
    y = y0
    n = 0
    while (n>=0):
        xmax, ymax = best_neighbor(s,f,x,y)
        n +=1
        if f(xmax,ymax) <= f(x,y):
            return x,y,n;
            break
        else:
            x = xmax
            y = ymax


# Run hill_climbing function from 100 random points in the range,
# returns a dataframe of the average and standard deviation of the
    number of
#steps to convergence and of the final value f*.

def hundred_points(s, f):
    arr = 10*np.random.rand(100,2)
    zl=[]
    nl=[]
    for p in arr:
        x,y,n = hill_climb(s,f,p[0],p[1])
        zl.append(f(x,y))
        nl.append(n)
    mean_= [mean(zl), mean(nl)]
    std = [stdev(zl),stdev(nl)]
    data = [mean_,std]
    df = pd.DataFrame(data)
    df = df.rename(index={0: "Average value", 1: "Standard
    deviation"},columns={0:"Final value f*", 1:"Number of steps to
    convergence"})
```

```
    return df

def standardDeviation(val, squaredVal, n):
   return np.sqrt((n * squaredVal - pow(val, 2)) / (n * (n - 1)))

print("\n f1: Results for a step size of 0.01")
print(hundred_points(0.01,f1))

print("\n f1 - Results for a step size of 0.05")
print(hundred_points(0.05,f1))

print("\n f1 - Results for a step size of 0.1")
print(hundred_points(0.1,f1))

print("\n f1 - Results for a step size of 0.2")
print(hundred_points(0.2,f1))

print("\n f2 - Results for a step size of 0.01")
print(hundred_points(0.01,f2))

print("\n f2 - Results for a step size of 0.05")
print(hundred_points(0.05,f2))

print("\n f2 - Results for a step size of 0.1")
print(hundred_points(0.1,f2))

print("\n f2 - Results for a step size of 0.2")
print(hundred_points(0.2,f2))
```





```
 f1: Results for a step size of 0.01
                    Final value f*  Number of steps to convergence
Average value             1.760513                      273.580000
Standard deviation        0.340032                      166.476569

 f1 - Results for a step size of 0.05
                    Final value f*  Number of steps to convergence
Average value             1.751941                       61.350000
Standard deviation        0.348362                       30.853638

 f1 - Results for a step size of 0.1
                    Final value f*  Number of steps to convergence
Average value             1.789323                       30.690000
Standard deviation        0.317804                       16.058413

 f1 - Results for a step size of 0.2
                    Final value f*  Number of steps to convergence
Average value             1.745745                        15.63000
Standard deviation        0.340060                         8.53354

 f2 - Results for a step size of 0.01
                    Final value f*  Number of steps to convergence
Average value             8.860000                      599.320000
Standard deviation        2.365663                      236.253625

 f2 - Results for a step size of 0.05
                    Final value f*  Number of steps to convergence
Average value             8.620000                      111.650000
Standard deviation        2.537716                       48.301113

 f2 - Results for a step size of 0.1
                    Final value f*  Number of steps to convergence
Average value             8.500000                       57.910000
Standard deviation        2.611165                       24.261661

 f2 - Results for a step size of 0.2
                    Final value f*  Number of steps to convergence
Average value             9.100000                       30.780000
Standard deviation        2.153222                       12.645581
```

We notice that a larger step size leads to a generally worse result. We obtain a higher value whenever our step sizes are smaller.

b) Repeat using local beam search with beam width in [2, 4, 8, 16], performing 100 runs of each. Were you able to improve performance over hill climbing for each function, as measured by the mean and standard deviation of the number of iterations and/or final objective function value?

```
#3 b)
def beamNeighbours(topK, stepSize, func, beamWidth):

    neighbours = [(0, 0, 0)] * (8 * beamWidth)
    i = 0

    for (x, y, val) in topK:
        neighbours[8 * i] = (x - stepSize, y - stepSize, func(x - stepSize, y - stepSize))
        neighbours[8 * i + 1] = (x - stepSize, y, func(x - stepSize, y))
        neighbours[8 * i + 2] = (x, y - stepSize, func(x, y - stepSize))
        neighbours[8 * i + 3] = (x + stepSize, y + stepSize, func(x + stepSize, y + stepSize))
        neighbours[8 * i + 4] = (x + stepSize, y, func(x + stepSize, y))
```

```
        neighbours[8 * i + 5] = (x, y + stepSize, func(x, y + stepSize))
        neighbours[8 * i + 6] = (x - stepSize, y + stepSize, func(x - stepSize, y + stepSize))
        neighbours[8 * i + 7] = (x + stepSize, y - stepSize, func(x + stepSize, y - stepSize))
        i += 1

#Remove out-of-bounds neighbours by setting their value very low
    i = 0
    for (x, y, val) in neighbours:
        if (x < 0) or (x > 10) or (y < 0) or (y > 10):
            neighbours[i] = (0, 0, -100)
        i += 1
    return neighbours


#Now implement the main beam algorithm
stepSize = 0.05
beams = [2, 4, 8, 16]


for func in [f1, f2]:
    df = pd.DataFrame()

    for beamWidth in beams:

        sum = 0; average = 0; squaredSum = 0; squaredAverage = 0

        #beams 100 times
        for i in range(0, 100):

            iteration = 0
            topK = [(0, 0, 0)] * beamWidth

            for j in range(0, beamWidth):

                topK[j] = [(10 * rand.uniform(0, 1), 10 * rand.uniform(0, 1), 0)]
                [(x, y, val)] = topK[j]; topK[j] = (x, y, func(x, y))

            #sort values in decreasing order
            topK = sorted(topK, key = itemgetter(2), reverse = True)

            #Min value of the topK list
            (xMin, yMin, topKMin) = topK[-1]

            #Generate neighbours and sort by values, largest first
            neighbours = sorted(beamNeighbours(topK, stepSize, func, beamWidth), key=itemgetter(2), reverse=True)

            #Max value of the neighbours
            (xMax, yMax, neighbourMax) = neighbours[0]

            #Keep top k values out of the topK and neighbours lists
```

```
            while (neighbourMax > topKMin):

                neighbourhoods = sorted(topK + neighbours, key=
    itemgetter(2), reverse=True)

                for j in range(0, beamWidth):

                    (xMax, yMax, neighbourMax) = neighbourhoods[j]
                    topK[j] = (xMax, yMax, neighbourMax)

                (xMin, yMin, topKMin) = topK[-1]
                neighbours = sorted(beamNeighbours(topK, stepSize,
    func, beamWidth), key=itemgetter(2), reverse=True)
                (xMax, yMax, neighbourMax) = neighbours[0]
                iteration += 1

            (xFinal, yFinal, best)= topK[0]


            average += best; sum += iteration
            squaredAverage += pow(best, 2); squaredSum += pow(
    iteration, 2)

        averageSteps = sum / 100; averageValue = average / 100
        standardDeviationSteps = standardDeviation(sum, squaredSum,
     100)
        standardDeviationValue = standardDeviation(average,
    squaredAverage, 100)
```

```
            Average Final Value  ...  Standard Deviation of Steps to Convergence
Beam Width                       ...
2.0                    1.767874  ...                                    31.085799
4.0                    1.828550  ...                                    28.999589
8.0                    1.883014  ...                                    18.592632
16.0                   1.953956  ...                                    13.832253

[4 rows x 4 columns]
            Average Final Value  ...  Standard Deviation of Steps to Convergence
Beam Width                       ...
2.0                    9.363690  ...                                    49.041597
4.0                    9.663800  ...                                    41.173915
8.0                    9.902158  ...                                    40.555362
16.0                   9.965400  ...                                    32.539176
```

We notice that th elarger the beamwidth, the better results we get for both the average final value and the standard deviation of the number of steps