Assignment 1 **LE, Nhat Hung**
COMP 424 - Artificial Intelligence McGill ID: 260793376
Prof. Jackie Chi Kit Cheung Date: February 1, 2019
Winter 2019 Due date: February 4, 2019

Question 1: Language Generation

We would like to design a simple language generation system which can generate sensible and grammatically correct sentences of English. 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 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 search problem, stating each of the parts of the search problem as shown in class.

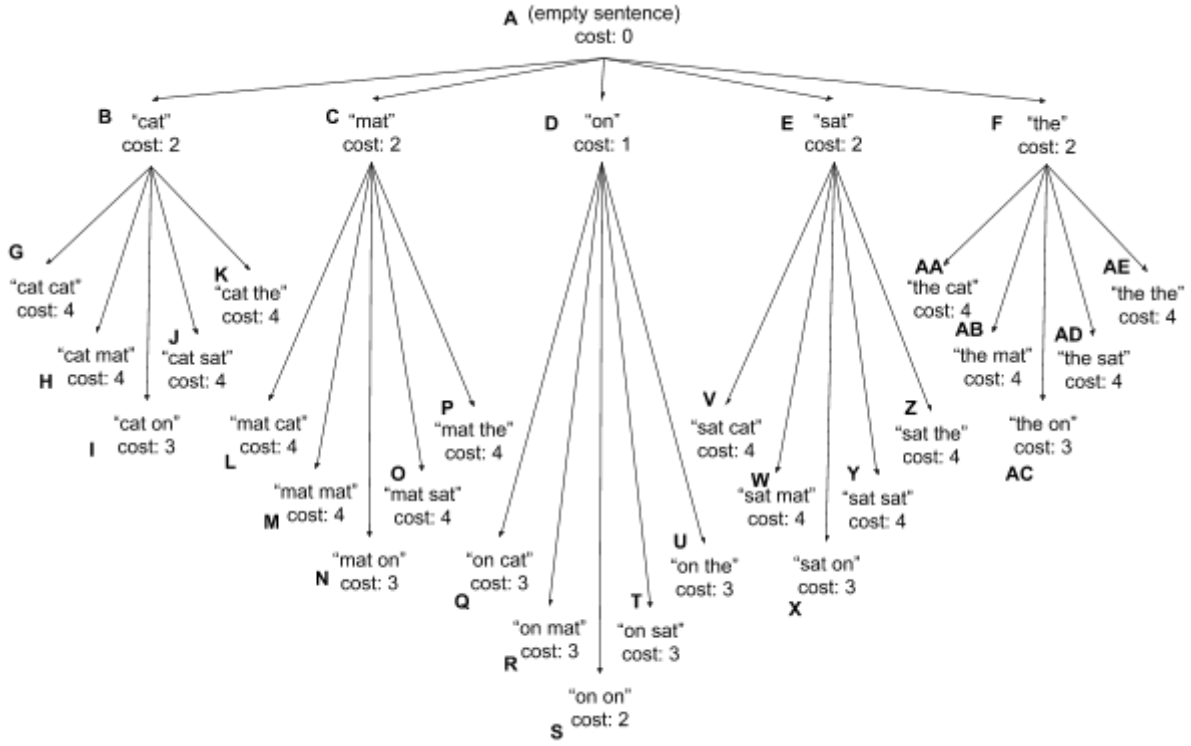
States: sentences of words from {“the”, “cat”, “sat”, “on”, “mat”}
Goals: {“the cat sat”, “the cat sat on mat”}
Operators: Adding a word from {“the”, “cat”, “sat”, “on”, “mat”} to sentence
Path cost: Number of consonants in generated sentence of current state

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.

The search graph is acyclic because no two states can arrive at the same state through an action, the action specifically being adding a word to the sentence represented by the current state.

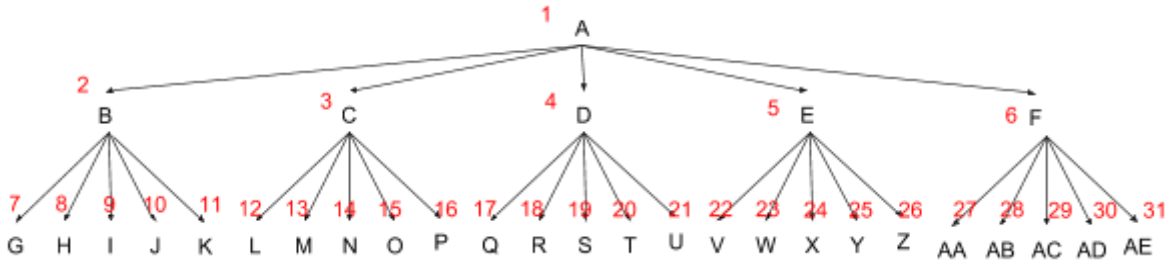
Therefore, the graph is similar to the the search tree illustrated in part c). It expands infinitely, each node having 5 outgoing edges representing the action of adding a word from {“the”, “cat”, “sat”, “on”, “mat”} with costs {2, 2, 2, 1, 2} respectively.

c) Draw the search tree down to a depth of 2 (i.e., the tree has 3 levels in total, including the root.)



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

i. Breadth first search



…

continue until goal node “the cat sat” is reached

ii. Uniform cost search

Priority queue = {(empty, 0)}

= {~~(empty, 0)~~, (D,1), (B,2), (C,2), (E,2), (F,2)}

= {~~(empty, 0), (D,1)~~, (B,2), (C,2), (E,2), (F,2), (S,2), (Q,3), (R,3), (T,3), (U,3)}

= {~~(empty, 0), (D,1), (B,2)~~, (C,2), (E,2), (F,2), (S,2), (I,3), (Q,3), (R,3), (T,3), (U,3), (G,4), (H,4), (J,4),

I has same priority as Q, R, T, U, but is before

(K,4)} them alphabetically
= {~~(empty, 0), (D,1), (B,2), (C,2)~~, (E,2), (F,2), (S,2), (I,3), (N,3), (Q,3), (R,3), (T,3), (U,3), (G,4), (H,4), (J,4), (K,4), (L,4), (M,4), (O,4), (P,4)}
= {~~(empty, 0), (D,1), (B,2), (C,2), (E,2)~~, (F,2), (S,2), (I,3), (N,3), (Q,3), (R,3), (T,3), (U,3), (X,3), (G,4), (H,4), (J,4), (K,4), (L,4), (M,4), (O,4), (P,4), (V,4), (W,4), (Y,4), (Z,4)}
= {~~(empty, 0), (D,1), (B,2), (C,2), (E,2), (F,2)~~, (S,2), (I,3), (N,3), (Q,3), (R,3), (T,3), (U,3), (X,3), (AC, 3), (G,4), (H,4), (J,4), (K,4), (L,4), (M,4), (O,4), (P,4), (V,4), (W,4), (Y,4), (Z,4), (AA,4), (AB,4), (AD,4), (AE,4)}
…

Repeat process of dequeuing a node, enqueueing its children then rearranging the queue to respect first the priority and alphabetical ordering. Continue until a goal node is reached.

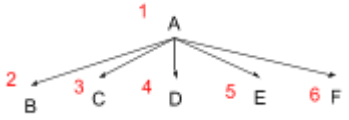
iii. Depth first search

In this case, DFS does not visit the goal node and infinitely expands: (empty), “cat”, “cat cat”, “cat cat cat”, “cat cat cat cat”, …

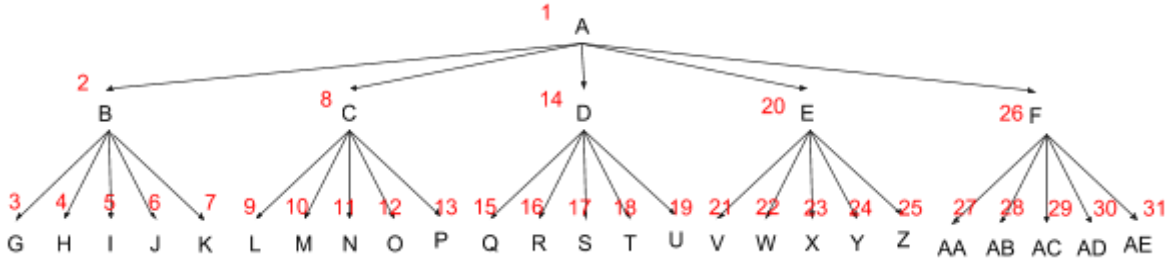
iv. Iterative deepening

limit = 0

limit = 1



limit = 2



…
At limit = 3, the goal node “the cat sat” will be visited.

Question 2: Search algorithms
(Adapted from Russell & Norvig)

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

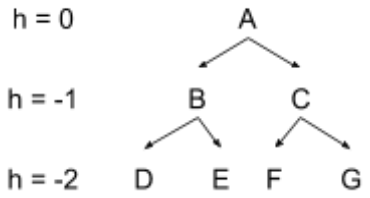
> Linear search tree i.e. each state only has one child state, with goal state at depth n.
> DFS takes $n = O(n)$ steps.
> IDS takes $1 + 2 + ... + n = O(n^2)$ steps.

Prove each of the following statements, or give a counterexample:
b) Breadth-first search is a special case of uniform-cost search.

> When all actions have the same cost, the priority queue used in uniform-cost search will never have to rearrange nodes that enter it (because a node's cost is proportional to its depth in the tree). The priority queue therefore acts no different to a simple queue, and uniform-cost search therefore acts no different to BFS. □

c) Depth-first search is a special case of best-first tree search.

> Let $h(n)$ the heuristic cost of node n.
> Consider the case where nodes at each depth share the same h value, and the lower the depth, the lower the h value e.g. $h(n) = -depth(n)$:



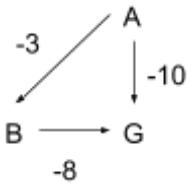
> Because best-first chooses lowest heuristic values first, nodes will be visited in the same order as in DFS. □

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

> A* greedily search with $f(n) = g(n) + h(n)$, $g(n)$ being the cost of the path so far.
> With $h(n) = 0 \forall n$, $f(n) = g(n)$, and A* will reproduce uniform-cost. □

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

> Counterexample:



Because of the restriction $h(n) \geq 0$, no admissible heuristic exists.
A* will thus go for suboptimal path A-G instead of A-B-G.
Therefore, A* is not optimal in the case where negative edge weights are allowed. □

Question 3: Optimization

Consider the following functions:

$f_1(x,y) = sin(x/2) + cos(2y)$

$f_2(x,y) = -|x-2| - |0.5y+1| + 3$

We would like to maximize these functions within the range of $0 \leq x,y \leq 10$. 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^*, f_2^* for 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?

f_1 **report:**

Step size	Mean f max	f max std deviation	Mean num steps	num steps std deviation
0.01	1.84326981335896	0.53144253647686	269.78	174.109539083876
0.05	1.88208935196088	0.46519173003823	52.79	33.7553832743756
0.1	1.90045543327599	0.42688375193690	28.82	17.0266731923767
0.2	1.87688317161356	0.46525753311700	13.7	7.97558775263616

Figure 1: Hill climbing with f_1

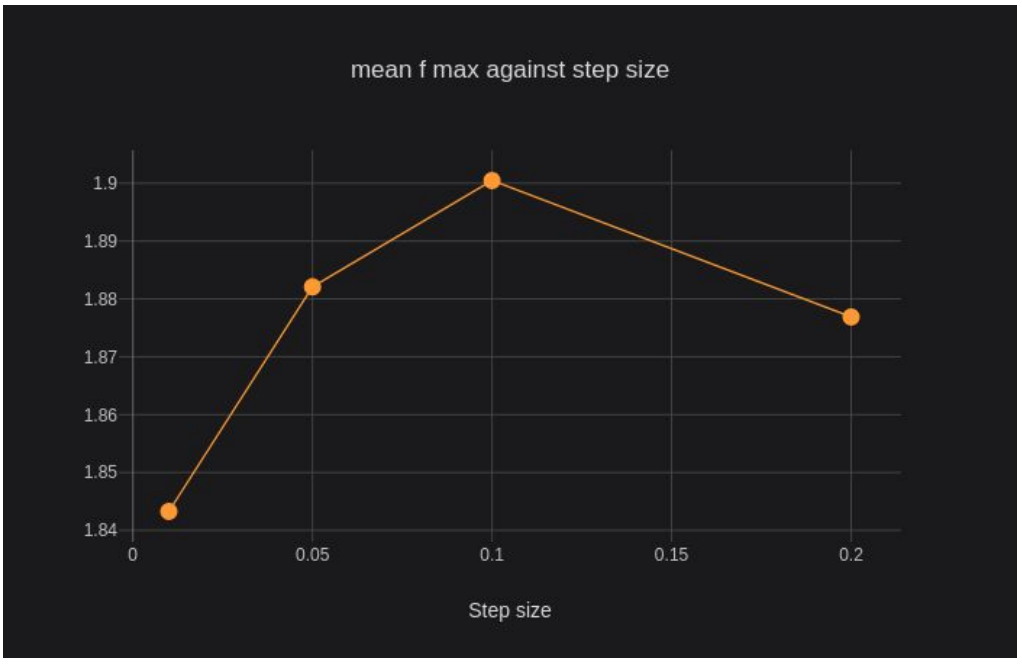


Figure 2: Mean f_1^* with varying step sizes. Target value is 2

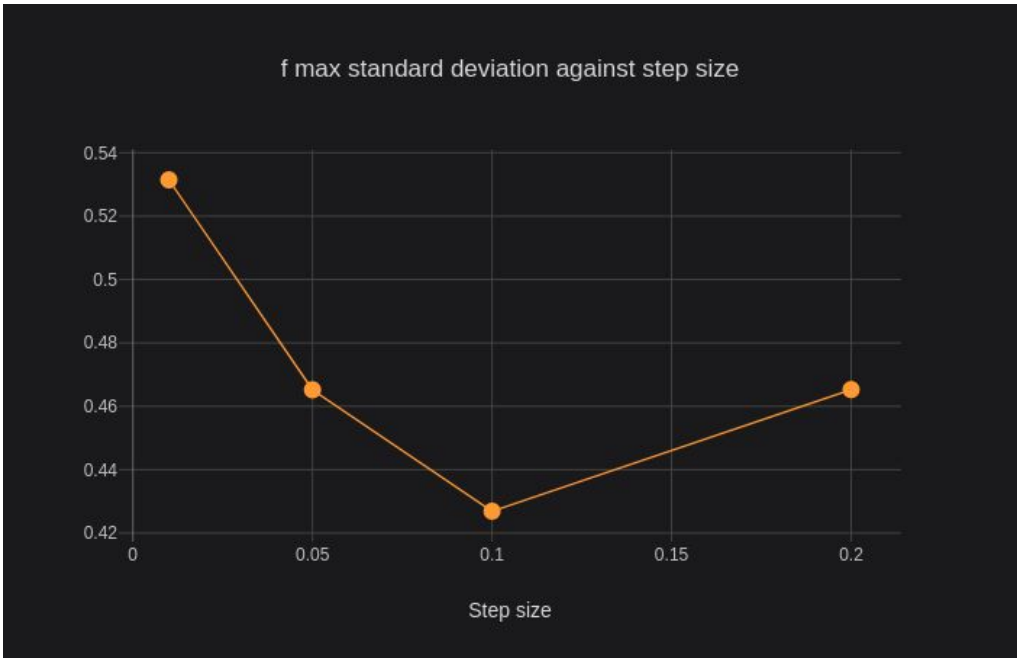


Figure 3: f_1^* standard deviation with varying step sizes

From above, we observe that as the step size increases, Hill Climbing's accuracy for f_1

- Is low at small step size: from above, highest standard deviation and lowest mean f_1^* at **step size = 0.01**.
- Reaches a maximum at **step size = 0.1**, where the mean is most near the target value of 2 and the standard deviation is minumum.

- After step size = 0.1, its accuracy decreases once again albeit at a slower rate: mean f_1^* decreases from 1.9 to 1.877 and standard deviation increases from 0.427 to 0.465 (fig. 1) .

Note that the runtime is also better at step size = 0.1 than at lower values: 28.82 compared to 52.79 (step size = 0.05) and 269.78 (step size = 0.01).

f_2 report:

Step size	Mean f max	f max std deviation	Mean num steps	num steps std deviation
0.01	1.99745918317189	0.00157599070793	633.28	225.510535452337
0.05	1.98805158662955	0.00736987754086	128.19	46.3453762526532
0.1	1.97388655902138	0.01517613298452	59.82	25.0676604412936
0.2	1.94628129779139	0.02848149594357	31	12.4354332453678

Figure 4: Hill climbing with f_2

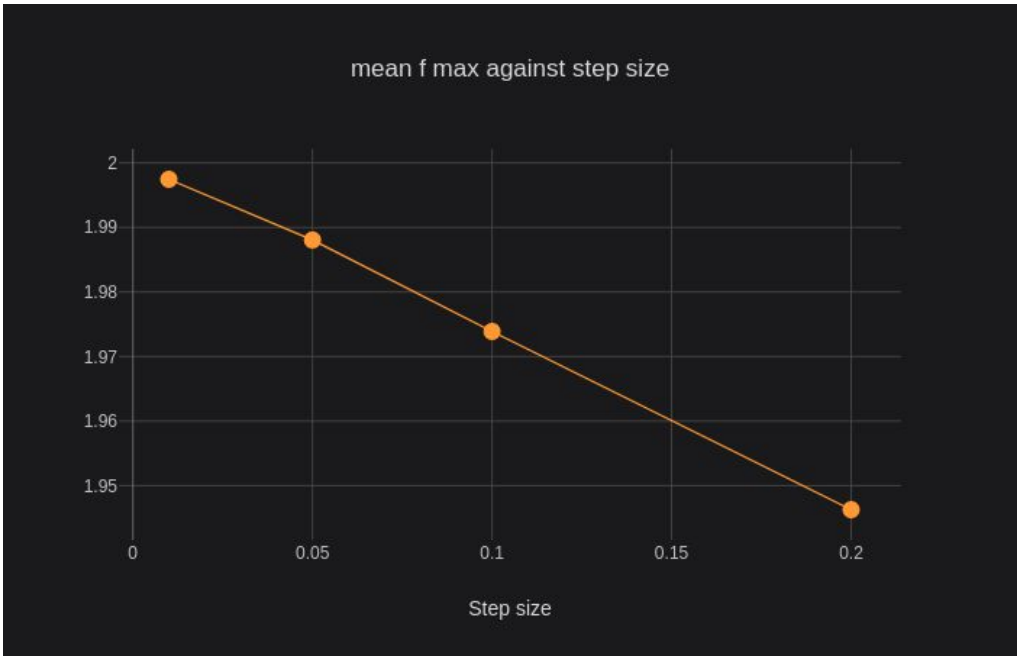


Figure 5: Mean f_2^* with varying step sizes. Target value is 3

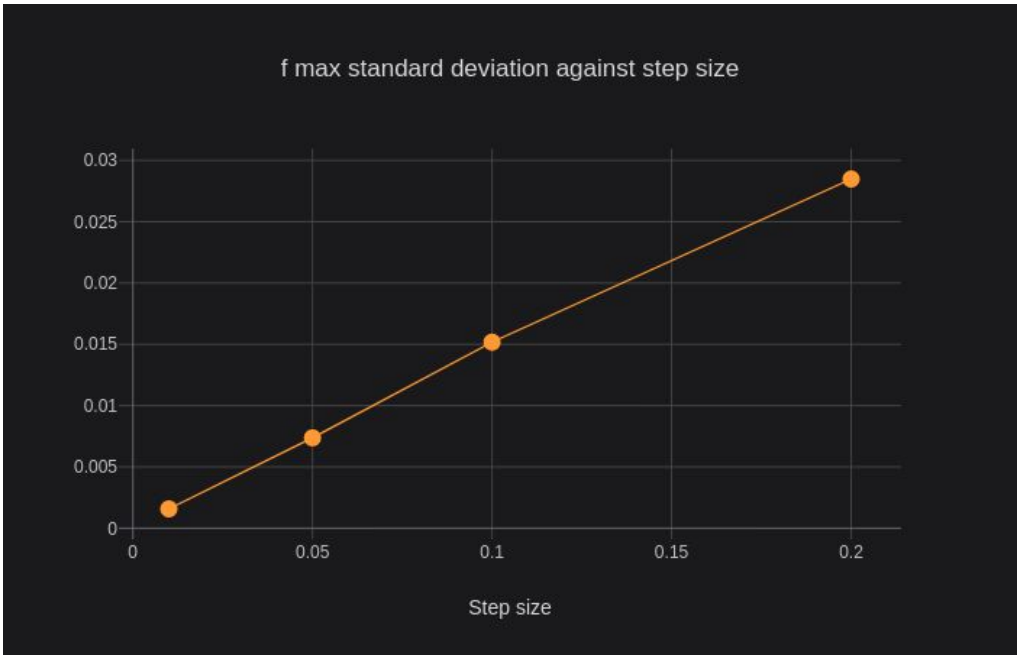


Figure 6: f_2^* standard deviation with varying step sizes

Hill climbing performs worse on f_2 as the step size increases. The mean of f_2^* decreases and its standard deviation increases linearly. Out of the given step sizes. 0.01 is the clear winner.

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?

From part a), we will choose 0.1 and 0.01 as the step sizes for f_1 and f_2 respectively.

f_1 report:

Beam width	Mean f max	f max std deviation	Mean num steps	num steps std deviation
2	1.95919962372069	0.27442426363670	22.52	15.3084813093918
4	1.99841598077104	0.00154874813457	16.03	10.4474446636486
8	1.99851309780026	0.00151190878197	12.22	6.34914167427377
16	1.99874973605555	0.00144307222893	9.67	4.98809582907145

Figure 7: Local beam search implementing hill climbing with f_1

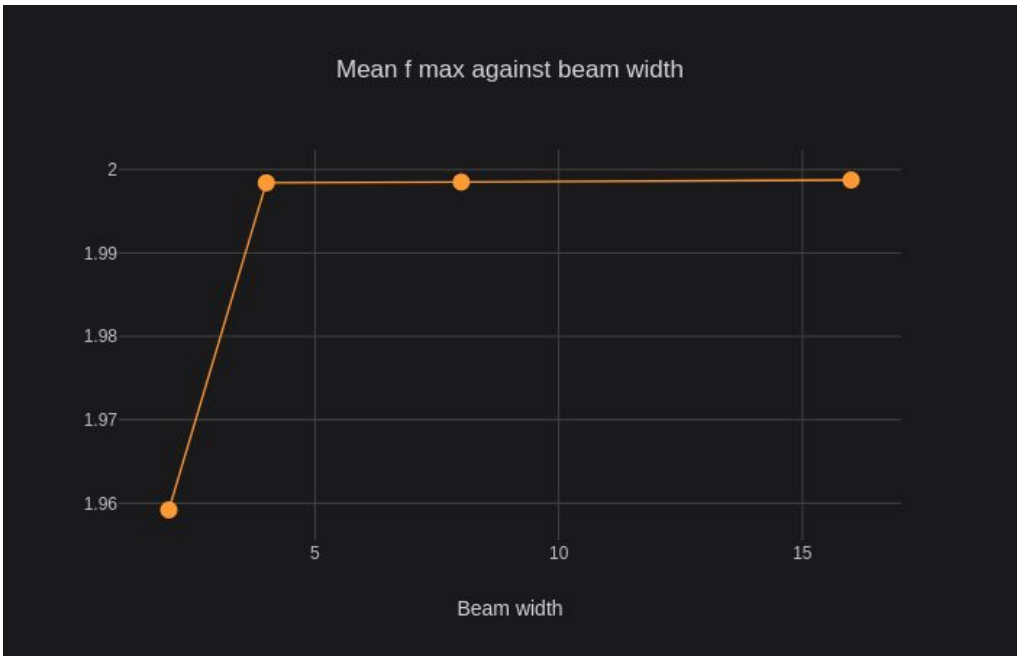


Figure 8: mean f_1^* with varying beam widths. Target value is 2

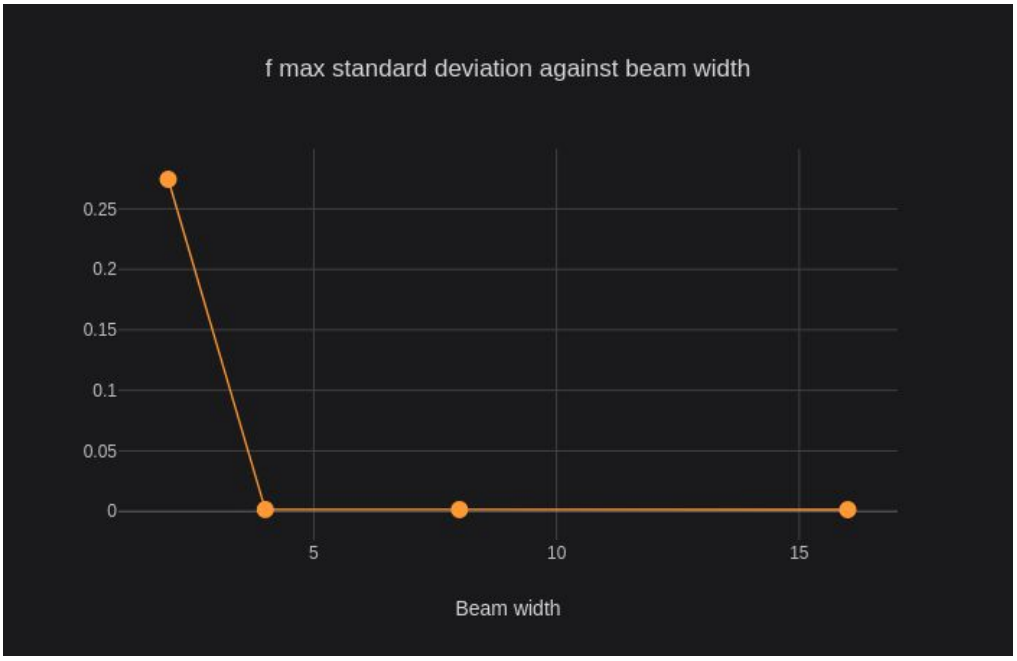


Figure 9: f_1^* standard deviation with varying beam widths

Unlike with step sizes, increasing the beam width only increases beam search's accuracy.

The mean f_1^* starts off at 1.959 (fig. 7), which is already higher than any final mean f_1^* obtained through hill climbing alone (fig. 1). It then only increases along with the beam width, reaching 1.9987 at beam width = 16, the closest to the target value 2 than with any other previous configurations.

Similarly, the standard deviation of f_1^* continuously decreases as the beam width increases, reaching its lowest point 0.0014 (fig. 7) at beam width = 16, which is lower than any other values obtained by hill climbing alone (fig. 1).

However, this improvement costs more in terms of runtime: the higher the improvement, the higher the number of iteration. The "mean num steps" column of fig. 7 describes the number of beams needed to reach f_1^* for each run. The real cost can then be calculated with:

$$\text{real cost} = \text{num steps} \times \text{beam width}$$

The lowest real cost from fig. 7 is $22.52 \times \textit{beam width} = 22.52 \times 2 = 45.04$, which is already higher than the number of iterations needed by hill climbing alone at the same step size of 0.1.

Overall, for f_1, local beam search is an improvement in accuracy over hill climbing alone, but at the cost of lower runtime.

f_2 report:

Beam width	Mean f max	f max std deviation	Mean num steps	num steps std deviation
2	1.99745206501296	0.00140355461288	477.38	226.834643738561
4	1.99759560895180	0.00146234151209	370.39	200.222321183228
8	1.99762777965194	0.00137987187349	259.54	167.413047281267
16	1.99753544492217	0.00141662046820	173.23	110.000441362750

Figure 10: Local beam search implementing hill climbing with f_2

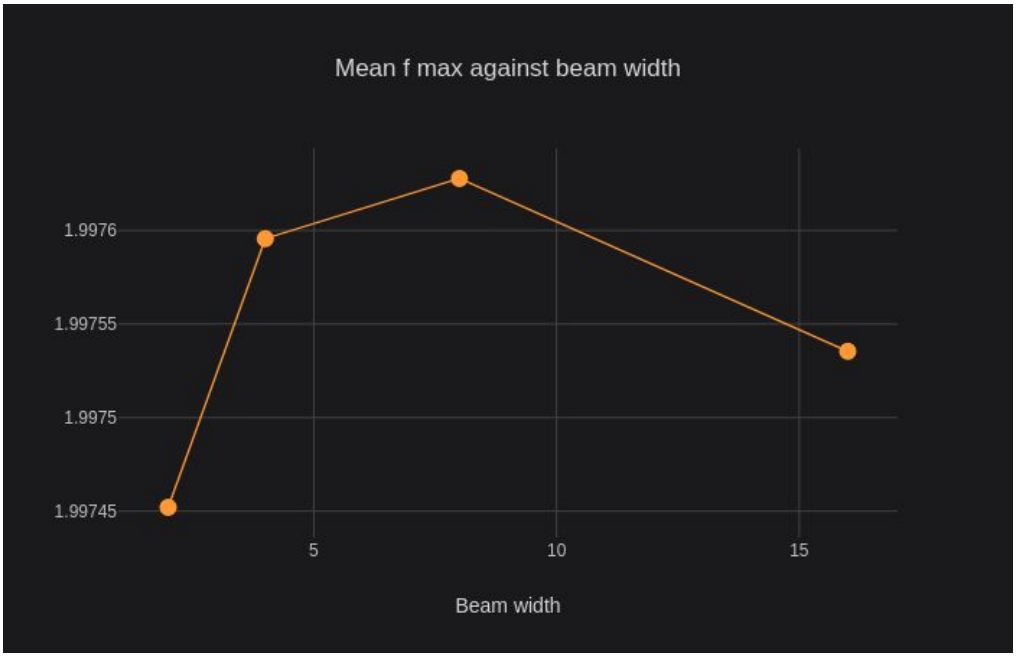


Figure 11: mean f_2^* with varying beam widths. Target value is 3

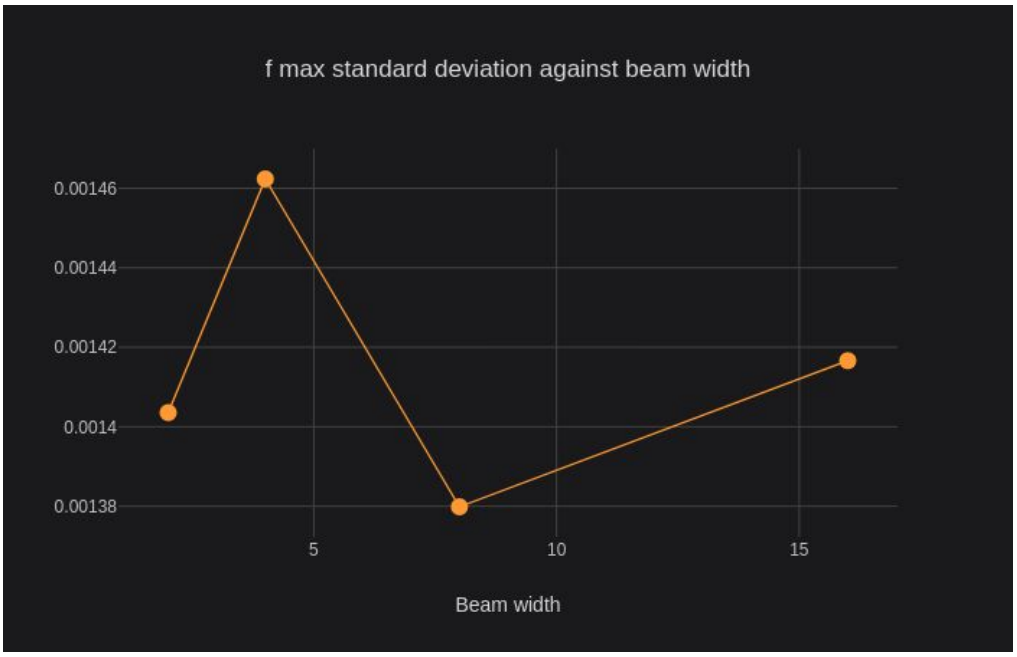


Figure 12: f_2^* standard deviation with varying beam widths

From above, beam width = 8 seems to be the best setting for local beam search on f_2. The mean and standard deviation of f_2^* behaves more unpredictably than with only hill climbing (which is only linear, fig. 5 and 6).

Beam search for f_2 doesn't deliver a real improvement. The highest achieved mean f_2^* is 1.9976 (fig. 10), which is more or less equivalent to hill climbing's 1.9975 (fig. 4). The same can be said about the two f_2^* standard deviations: 0.0014 (beam search) and 0.0016 (hill climbing).

Moreover, the mean number of iterations needed to obtain the optimal f_2^* in local beam search far exceeds what hill climbing would need: $259.54 \times 8 = 2076.32$ (beam width = 8, fig. 10) as opposed to 633.28 (step size = 0.01, fig. 4).

Therefore, applying local beam search on f_2 is not a performance improvement over hill climbing.