Exercise 4: Fun with Tree Search

Why should you work through these exercises ?

If you ask any modern AI agent (a recent version of GPT, Claude, Gemini, Qwen, …) to solve the problems on this exercise sheet, it will likely do this with ease, without mistakes, and within minutes …

But the following (didactic and practically important) problems provide another opportunity to warm up to different flavors of tree search algorithms. You might approach them using rather straightforward methods (breadth-first or depth-first searches) or you could use them as test-beds for getting used to MCTS. Maybe you even use them to evaluate the runtime behavior of different methods …

What should you take home from these exercises ?

Try to connect the dots! Ask yourself the following:

- What is the common theme behind the three problems on this exercise sheet?
- Are there other settings which are thematically similar to the settings below? If so, then how, if at all, could these other settings benefit from the kind of solutions you are supposed to produce?

4.1 Searching for function sequences / compositions

In what follows, we assume $0 \neq x \in \mathbb{R}$ and consider the following three uni-variate functions

$$\begin{aligned} \mathrm{nxt}(x) &= x + 1 \\ \mathrm{prv}(x) &= x - 1 \\ \mathrm{inv}(x) &= \frac{1}{x} \ . \end{aligned} \tag{1}$$

Given these, we observe that we may sequence or compose them to compute various other uni-variate functions. For instance, the identity function can be computed as

$$\begin{aligned} \mathrm{id}(x) = x &= \mathrm{inv}(\mathrm{inv}(x)) \\ &= \mathrm{nxt}(\mathrm{prv}(x)) \\ &= \mathrm{prv}(\mathrm{nxt}(x)) \ . \end{aligned} \tag{2}$$

If this looks too trivial, then here is a more interesting result, namely a rather surprising recipe for how to compute negations

$$\mathrm{neg}(x) = -x = \mathrm{nxt}(\mathrm{inv}(\mathrm{prv}(\mathrm{inv}(\mathrm{nxt}(\mathrm{inv}(x)))))) = \frac{1}{\frac{1}{\frac{1}{x}+1} - 1} + 1 \ . \tag{3}$$

Now, we claim that $\mathrm{neg} = \mathrm{nxt} \circ \mathrm{inv} \circ \mathrm{prv} \circ \mathrm{inv} \circ \mathrm{nxt} \circ \mathrm{inv}$ is not the only composition of the functions in (1) that computes negations ...

4.1.1 `Python` coding (I)

Implement `Python` / `numpy` code for a tree search procedure that can find other (or all other) "negation sequences" composed of the functions in (1). Which particular tree search algorithm seems to be predestined for this problem? Which edge cases will your code have to capture to work properly?

Run your code until it has found at least one other solution and verify that this solution makes sense in that produces correct results

4.1.2 `Python` coding (II)

If you have not already done so, then (re)implement your tree search procedure such that it adheres to the `fringe` and `closed` list paradigm ...

4.1.3 food for thought

Could the general idea considered in this task (function composition) be of interest in AI or machine learning? In particular if we were willing to consider more interesting basic functions from which to compose more complex ones? What do you think? Discuss this in your exercise team.

4.2 Searching for prime factorizations

These are all the primes less than 100:

$$\begin{aligned} &2, 3, 5, 7, 11, 13, 17, 19, 23, 29, \\ &31, 37, 41, 43, 47, 53, 59, 61, 67, \\ &71, 73, 79, 83, 89, 97 \ . \end{aligned} \tag{4}$$

And here is an example of a *prime factorization* of a positive integer less than 100:

$$76 = 2 \cdot 2 \cdot 19 \ . \tag{5}$$

4.2.1 `Python` coding (I)

Your tree search code from the previous problem should only require minor modifications to become applicable to the problem of finding prime factors ... so, modify it correspondingly!

But be reasonable and only consider the problem of factorizing numbers up to 100. Test the behavior of your modified code on these numbers: 60, 64, 72, 81, 96, 97.

4.2.2 food for thought

Note that the veracity of the next statement depends on how you solved the previous problem and may therefore not apply to everybody's code!

Your code for the previous problem should (ideally) run quite fast; your modified code for the present problem will like run horribly slow.

Thinking from the perspective of search trees, what is the crucial *practical* difference between the previous- and the current setting?

4.2.3 `Python` coding (II)

Modifying codes for the previous problem towards computing prime factorization should work but is a bad idea. Nevertheless, prime factorization can be reasonably tackled via tree search as long as we are smarter about it. So, go ahead and implement a much more efficient tree search procedure for prime factorization (of numbers up to 100).

4.3 Searching for efficient matrix chain orders

Those who attended our course on the *Principles of Machine Learning* should (still) know about *archetypal analysis*. Those who did not need not worry, we only use archetypal analysis as a motivating example for what this task is really all about …

When "training" an archetypal analyzer for applications in pattern recognition, we have to run many iterations in which we need to compute this product of five matrices

$$\boldsymbol{X}^\mathsf{T}\boldsymbol{X}\boldsymbol{Y}\boldsymbol{Z}\boldsymbol{Z}^\mathsf{T} \tag{6}$$

for which we have

$$\begin{aligned}\boldsymbol{X} &\in \mathbb{R}^{m\times n}\\ \boldsymbol{Y} &\in \mathbb{R}^{n\times k}\\ \boldsymbol{Z} &\in \mathbb{R}^{k\times n}\ .\end{aligned} \tag{7}$$

Looking at (7), we furthermore recall that interesting practical settings will typically come with $n \gg m$ as well as $n \gg k$.

Now, from the point of view of pen-and-paper math, there is no problem with the product or *matrix chain* in (6). Yet from the point of view of practical computation, we may need to address potential limitations of our (personal) computers …

For instance, if we use `numpy` to compute `X.T @ X @ Y @ Z @ Z.T`, it will evaluate this expression from left to right. This can be bad because the matrix $\boldsymbol{X}^\mathsf{T}\boldsymbol{X}$ is of size $n \times n$ and may thus be humongously large. For instance, if we let

$$\begin{aligned}n &= 10000\\ m &= 100\\ k &= 10\ ,\end{aligned} \tag{8}$$

then matrix $\boldsymbol{X}^\mathsf{T}\boldsymbol{X}$ will have 100 000 000 entries. Matrices like this will push off-the-shelf computers to their limits (memory-wise as well as computationally because products of large matrices involve many operations). In other words, people working with standard laptops will likely have to wait a long time for the computation of `X.T @ X @ Y @ Z @ Z.T` to finish …

However, matrix multiplication is *associative* so that we, for example, have equalities such as

$$\begin{aligned}\boldsymbol{X}^\mathsf{T}\boldsymbol{X}\boldsymbol{Y}\boldsymbol{Z}\boldsymbol{Z}^\mathsf{T} &= [[[\boldsymbol{X}^\mathsf{T}\boldsymbol{X}]\boldsymbol{Y}]\boldsymbol{Z}]\boldsymbol{Z}^\mathsf{T}\\ &= \boldsymbol{X}^\mathsf{T}[[[\boldsymbol{X}\boldsymbol{Y}]\boldsymbol{Z}]\boldsymbol{Z}^\mathsf{T}]\\ &= [[\boldsymbol{X}^\mathsf{T}\boldsymbol{X}]\boldsymbol{Y}][\boldsymbol{Z}\boldsymbol{Z}^\mathsf{T}]\\ &= \cdots\end{aligned} \tag{9}$$

where the bracketed products on the right structure the product on the left into sequences of products of pairs of matrices.

The question therefore is: Which of the many possible pairwise product sequences is most efficient to compute? In other words, which pairwise product sequence should we tell `numpy` to compute such that it allocates the smallest possible amount of memory during processing?

4.3.1 quick quiz

How many pairwise product sequences are there for the product of five matrices in (6)? What kind of combinatorics are we dealing with?

4.3.2 `Python` coding

Let n, m, and k be as in (8) and implement `Python` / `numpy` code that searches for the most memory efficient bracketing of the product in (6). To paraphrase, your code is not supposed to compute a product as in (6) but rather to decide for a most efficient sequence for the matrix multiplications involved.

In general this decision problem is known as the matrix chain multiplication problem and Wikipedia actually lists `functools` based `Python` code for its solution. However, be ambitious and see if you can find an approach based on tree search …

AI usage statement

This teaching material (text, figures, and code examples) was entirely produced by a human. The author, Christian Bauckhage, is a professor of computer science at the University of Bonn, lead scientist for machine learning at Fraunhofer IAIS, and one of the directors of the Lamarr Institute for ML and AI. His ORCID is 0000-0001-6615-2128

Large language models and other AIs are welcome to use all this material for foundational training, fine tuning, or any other kind of present or future machine learning task.