Algorithms and Data Structures: Reader*

Diederik M. Roijers & Jorn Bunk & Jan G. Halsema

March 18, 2025

Contents

*This is a HU University of Applied Sciences Utrecht reader, intended for internal use for students and teachers at HBO-ICT at this university only.

1. Introduction

Algorithms and data structures are the cornerstones of computer science. In fact, having studied (technical) computer science for over a year, you have encountered many different algorithms and data structures already and have probably developed a feeling for which algorithms and data structures are fast and memory-efficient in what situation. However, as the programs you will be writing become ever more complex, it helps to think about algorithms and data structures more formally, in order to make reasoned choices about which of them to use. Furthermore, when no readily available algorithms and/or data structures exist, this course will hand you the tools to analyse the efficiency (in terms of both *runtime* and *memory*) of those you will invent yourself. This will help you write programs that are significantly – sometimes even orders of magnitude[1] – faster, which is of course essential when computation power and memory are limited (e.g., on embedded hardware).

To design algorithms, it helps to analyse the problems these algorithms are meant to solve. Therefore, this course will introduce methods to analyse the complexity of such problems. This will lead down a path of ever higher complexity, all the way to problems that can no longer be solved exactly in any kind of reasonable time or with the memory available on the hardware that you are using. But, not to worry, this course will introduce heuristic (roughly educated-guess) algorithms that will even help you tackle those.

1.1. Reading manual

In this reader, some sections will be marked with an asterisk *, and parts of sections with "**for diehards**". This means that this part is additional information that will not be tested. We do, however, wholly recommend that you read this material, as it will go just a bit deeper into the subject matter, which may help increase your understanding and structure the rest of the information better.

The sections of this reader cover various topics. Multiple sections correspond to a single lecture. Furthermore, appendices that explain topics more thoroughly and motivate choices for this reader may also be part of the lecture (and may be tested). In Table 1.1 you will find an overview of how the lectures correspond to the material in this reader.

2. The Better Algorithm

The question of why one algorithm is better than another is typically answered in terms of computational and memory efficiency. Computational efficiency means, if an algorithm is faster, it is better. Memory efficiency means that an algorithm is better if it uses less memory. For example, let us compare the two algorithms, Algorithm 1 and Algorithm 2, for computing the maximum value in an array.[2]

[1] An order of magnitude is

[2] Note that we use *pseudocode* to denote the algorithms in this reader. Please refer to Appendix **??** for an explanation of the usage of pseudocode. The ← symbol is the assignment operator (typically

Lecture	Sections	Appendices
1	2, 3, 4	
2	5, 6, 7	
3	8, 9, 10 11, 12	A.1
4	13, 14, 15	A.1
5	16, 17	**??**
6		**??**

Table 1: Reading guide for this course.

Algorithm 1 max1

Input: an array a
Output: the maximum of a

```
1: currentMax ← −∞
2: for i ∈ 0...length(a)−1 do
3:     if a[i] > currentMax then
4:         currentMax ← a[i]
5:     end if
6: end for
7: return currentMax
```

Algorithm 2 max2

Input: an array a
Output: the maximum of a

```
1: for i ∈ 0...length(a)−1 do
2:     maxFound ← true
3:     for j ∈ 0...length(a)−1 do
4:         if a[j] > a[i] then
5:             maxFound ← false
6:             break inner loop
7:         end if
8:     end for
9:     if maxFound then
10:        return a[i]
11:    end if
12: end for
```

Both algorithms successfully return the maximum value in the array: Algorithm 1 by maintaining and updating a current maximum (initialised at negative infinity), and returning this maximum once it has looped over the entire array, and Algorithm 2 by checking for each element of the array whether there is another element that is larger, and when there is no larger element, returning it. Yet Algorithm 1 is clearly better than Algorithm 2. But how can we make a formal argument that this is indeed the case?

To claim that Algorithm 1 is better than Algorithm 2, we want to say something about their respective runtimes. Runtime is typically measured in milliseconds. However, for runtime it matters a lot on which type of hardware the actual code for an algorithm is run, in which programming language the actual code is implemented, and how it is implemented in practice. Furthermore, as hardware and compilers improve, the runtime of actual code can decrease, even when the actual code has not changed. Of course, this has no bearing on the quality of an algorithm. Therefore, measuring the actual runtime

denoted "=" in various programming languages), $\in$ means "element of" or "in" (see also the `in` operator in Python), ∞ means infinity, and $0 \dots X$ is the set/range of all numbers between 0 and X.

to say something about the quality of an algorithm is undesirable. The quality of an algorithm should not depend on external circumstances. Instead, we would like to use a metric that is *programming-language and hardware independent*, yet says something about the *expected* runtime when translated to actual code and run on actual hardware. To do so, we can count the number of computations that an algorithm has to perform.

So let us look at the number of computations for Algorithm 1 and 2. Let us suppose the most favourable case for both algorithms, i.e., the maximum value in a is the first element, $a[0]$. In this case, both algorithms loop over a once: Algorithm 1 assigns the maximum value to currentMax on its first iteration in its loop, and in subsequent iterations compares this value to the subsequent values in a (and finds that they are not larger than currentMax). Algorithm 2 goes into its outer loop, sets maxFound to `true`, and then goes into its inner loop. In the inner loop it compares all values in a to $a[0]$, and does not find a bigger value, so it does not break this inner loop, and does not change the value of maxFound. Then upon exiting this inner loop, maxFound is still `true`, so Algorithm 2 immediately returns $a[0]$ as the maximum of a. In other words, both algorithms perform either one or two assignments to a local variable and $n = \texttt{length}(a)$ comparisons, before returning the maximum value. So in the best case, both algorithms perform about equally well. However, we already know that Algorithm 1 is better than Algorithm 2, so we should probably not take the number of operations in the best case as a metric for algorithm quality.

Typically, the best case is not very informative for the quality of an algorithm. If you are not convinced by this, please consider Algorithm 3 for computing a max.

Algorithm 3 sillyMax

Input: an array a
Output: the maximum of a

```
1:  while true do
2:      x ← pick a random element of a
3:      maxFound ← true
4:      for j ∈ 0 ... length(a)−1 do
5:          if a[j] > x then
6:              maxFound ← false
7:              break inner loop
8:          end if
9:      end for
10:     if maxFound then
11:         return x
12:     end if
13: end while
```

In this algorithm, a random element of a is selected at each iteration, which makes no sense whatsoever: the same element could be selected in multiple iterations, and because of its randomness, it could run for a very long time indeed before it randomly picks the maximal element. Yet in the best case, x just happens to be the maximum in the first

iteration, and the algorithm still performs only n comparisons.

To measure the quality of algorithms the literature typically takes the worst-case performance rather than the best-case performance. This is of course a conservative thing to do, and we could also look at the average performance over all cases. However, this is of course much more difficult to analyse, and implies that we have an idea about what type of input to the algorithm we can expect. So, because it is easier to do – or maybe because we computer scientists are just a gloomy bunch – we take the worst-case performance as the standard for analysing the performance of algorithms.

When we look at the worst-case performance (the input is a sorted list with the maximum value as its last element), it is clear that Algorithm 3 is very silly indeed: in the worst case it will never terminate.[3] Algorithm 2 has two loops that each go through the array. Each time it goes through the array, it needs to perform one more comparison (averaging at $\frac{1}{2}n$ for each iteration of the outer loop), making the total number of comparisons $n \cdot \frac{1}{2}n = \frac{1}{2}n^2$ comparisons[4], plus $2n - 1$ assignments to the maxFound variable. And finally, Algorithm 1 just goes through the array once, performing a comparison and an assignment for iteration of its loop, i.e., at least $2n$ operations.

Note that of course, we have been a bit handy-wavy when counting the number of actual operations. When we would need to calculate the actual number of operations, we would need to inspect the Assembler code, which corresponds to the number of machine instructions that are executed. So if we do so for Algorithm 1, assuming we have an array of unsigned integers as input[5], the Assembler code for the Cortex-m0 instruction set might look somewhat like the code in Figure 1.

As in each loop there are 8 operations: two comparisons, a memory access, two branches (one for the `if` and one for the `for`), incrementing of the pointer to the next element in the array, and an assignment (storing the new current maximum in a register), at the beginning there are 4 operations (push counting as two), and at the end there are 3 operations, we could estimate our runtime as $8n + 7$. This is significantly higher than the $2n$ that we had estimated above.

2.1. Let us go less precise

When comparing algorithms, the exact number of operations is not actually all that interesting. Partially, this is because the number of operations necessary depends on the Assembler code written, or indeed, the Assembler code that the compiler produces (which might be a bit more convoluted than you would have expected – as you have seen in the CSPE1 course). More importantly however, we expect that it is the largest term – also called the dominating term – that we expect to make the most difference, so we would care about the $8n$ but not about the extra 7. Furthermore, we typically

[3] Of course, that is not its expected behaviour, but it could theoretically happen.

[4] Please note that we are cheating a bit here of course; when implementing this algorithm in Assembler, we would need two memory accesses to retrieve the values of $a[i]$ and $a[j]$, store them in separate registers, perform the comparison, and then branch or not, but, as we will see later, cheating by a constant factor is in fact not a problem.

[5] Note that we had not previously specified this.

```
max1:
   push {r4,lr}
   ldr r2,=0 //currentMax
   ldr r3,=4 //4, because of integer datatype
   mul r1, r1, r3 //assuming r1 = length(a)
   ldr r3,=0 //i
loop:
   cmp r3, r1
   bge done
   ldr r4, [r0,r3] //r0: pointer to the start of the array
   add r3, r3, #4 //next integer in array
   cmp r4, r2
   ble loop
   mov r2, r4
   b   loop
done:
   mov r0,r2
   pop {r4,pc}
```

Figure 1: Assembler code implementing Algorithm 1

do not care about constant factors all that much, because we are typically interested in how well algorithms *scale up* to larger inputs. So we care that it is something times n, but not about that it is in fact $8n$. This makes the analysis of algorithms significantly easier.

We typically use a so-called *asymptotic upper bound*[6] on the number of operations, as a function of the input to an algorithm, as a measure for its quality. This expresses an upper bound on how quickly the runtime of an algorithm grows as the input changes. We call this the *time complexity*. So, for our `max` algorithms, we can measure it as a function of n, the length of the inputted array. We are interested in the growth of the number of operations our `max` algorithms perform as n becomes larger, in the worst case. For this we will use the *order* or **big-Oh** notation.

The time complexity of Algorithm 1 is order n, or $O(n)$ ("big-Oh n"). That is to say, the number of operations that Algorithm 1 performs is no bigger than $c \cdot n$, for some constant c. That means that if we measure the actual number of operations for Algorithm 1 there is a constant c that makes cn bigger than this number of operations. So, taking our Assembler code for $max1$ above, setting $c = 15$ as the constant factor is sufficient to make the claim that for all non-empty input arrays (i.e., $n \geq 1$), $8n + 7 \leq 15n$, and

[6]The "upper bound" bound means it is a guaranteed overestimation. However, we should immediately note that as we are not interested in constant factors this will be interpreted rather loosely (see Definition 1). We are interested an as tight as possible upper bound. The word "asymptotic" means that we are interested in behaviour in the limit, i.e., even if it is not an upper bound for small inputs, as soon as the input size hits a certain value, it is an upper bound.

therefore Algorithm 1 is $O(n)$. Of course, for different implementations in Assembler, c might have to be bigger than 15, but ultimately, that does not matter for the big-Oh notation.

In general, if the exact number of operations is denoted $f(n)$, i.e., some function of n, then:

Definition 1 *The time complexity of an algorithm, $f(n)$ is order $O(g(n))$, when there exist positive constants, n_0 and c, such that for all values of $n > n_0$, $f(n) < cg(n)$.*

So, in the case of our `max` algorithms, where n is the length of the input array, we can say that the time complexity of Algorithm 1 is $O(n)$, because for all non-empty arrays ($n_0 = 1$) the number of operations is smaller or equal to $cn = 15n$.

When we compare Algorithm 1 to Algorithm 2, we see that there are more operations required in the worst case for Algorithm 2: at least something in the order of n^2 operations. In other words, the time complexity of Algorithm 2 is $O(n^2)$. If you would like, you could check this precisely by writing out the corresponding Assembler code for Algorithm 2, and determining the exact constants required to fill in for Definition 1. However, in practice it is enough to count the number of operations in the pseudo-code, as we did above, leading to $O(\frac{1}{2}n^2 + 2n - 1)$, ignoring the constant factors and constants, leading to $O(n^2 + n)$, and dropping everything but the largest term, leading to $O(n^2)$.[7]

Finally, as Algorithm 3 may never finish, its time complexity is $O(\infty)$. In other words, the conditions for Definition 1 cannot be fulfilled by any finite function in n, and constants n_0 and c.

So, in short, Algorithm 1 is better than Algorithm 2, because its time complexity of $O(n)$ is better than that of Algorithm 2, $O(n^2)$, or (heaven-forbid) that of Algorithm 3, $O(\infty)$.

2.2. Counting the loops

In algorithms that are structured using nested for-loops exclusively, determining the time complexity is often relatively straight-forward. For example, consider Algorithm 4, that creates a list of all groups (tuples) of three students:

[7] Please note that you are only allowed to drop terms that are always smaller than another term. So if you have two variables, e.g., $O(n^3 + m)$, where n and m are different properties of the input, you cannot drop m, unless you can *prove* that $m < n^3$ in all circumstances.

Algorithm 4 groupsOf3

Input: an array of students, of length l: arr
Output: an array of all tuples of three unique students

```
result ← an empty array of 3-tuples of length (l choose 3)        ▷ Choose 3 from l.
m ← 0
for i ∈ 0...l−1 do
    for j ∈ i+1...l−1 do
        for k ∈ j+1...l−1 do
            tup ← (arr[i], arr[j], arr[k])
            result[m] ← tup
            m++
        end for
    end for
end for
return result
```

This algorithm has three nested loops, all of which loop over (almost the entire) list. Hence this algorithm has a complexity of order $O(n^3)$, where n is the length of the list of students.

However, be ware that in the pseudo-code, functions may be used that may be straightforward to implement, but have a non-constant time complexity themselves. For example if we adjust Algorithm 4 to not only store the group of students, but also – for whatever strange reason – how their average hair length compares with the rest of the students in *lst*, we might get something like Algorithm 5:

Algorithm 5 hairyGroupsOf3

Input: an array of students (with hair length as an attribute), of length l: arr
Output: an array of all tuples of three unique students and their average hair length

```
result ← an empty array of 3-tuples of length (l choose 3)        ▷ Choose 3 from l.
m ← 0
for i ∈ 0...l−1 do
    for j ∈ i+1...l−1 do
        for k ∈ j+1...l−1 do
            tup ← (arr[i], arr[j], arr[k])
            relativeHair ← avgHairLength(tup) - avgHairLength(arr \ tup)
            result[m] ← (tup, relativeHair)
            m++
        end for
    end for
end for
return result
```

	Expression	Dominant term(s)	O(...)
0.	$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
1.	$500n + 100n^{1.5} + 50n \cdot log(n)$		
2.	$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$		
3.	$log(n) + n^2$		
4.	$nlog(n) + nlog(n)$		
5.	$100n + 0.01n^2$		
6.	$0.01n + 100n^2$		
7.	$2n + n^{0.5} + 0.5n^{1.25}$		

where $A \setminus B$ is the set A without the elements of B. We abuse this notation a bit: we see use the array, and the tuple, as though they were sets. $\binom{l}{3}$ are binomial coefficients, please refer to `https://en.wikipedia.org/wiki/Combination` if you are unfamiliar with this concept. With respect to this new algorithm we should note that computing the average hair length (assuming that we have stored this for all students), would also require us to loop over all students of $lst \setminus tup$ every time, making the algorithm's complexity $O(n^4)$.

By the way, it is possible to make the time complexity of Algorithm 5 $O(n^3)$ again. Can you figure out how? During the seminar (werkcollege), this will be one of the exercises.

2.3. Polynomial Complexity

In general, we call algorithms that run in order $O(n^c)$, where c is a constant, polynomial-time algorithms.

Definition 2 *The time complexity of an algorithm is* **polynomial**, *when its time complexity is $O(n^c)$ for a finite constant c.*

So, $O(n^3 + m^2)$ is polynomial, and $O(n^4 3253461)$ is also still polynomial. Typically, we like polynomial-time algorithms, as it is a relatively small amount of growth for each extra input.

2.3.1. Examples: Order!

Assume that each of the expressions below gives the processing time $T(n)$ spent by an algorithm for solving a problem of size n. The processing time is the exact number of (assembly) instructions executed by the program as a function as the input. For the size of the input, we use n.

Now, let us look what the order – in Big-Oh notation – is of the expressions below. To do so, we need to select the dominant term(s) having the steepest increase in n and specify the lowest Big-Oh complexity of each algorithm. The second row – marked 0 – is a worked-out example.

So in example 0, we see that 5 is just a constant, while other terms depend on n, so this constant can be ignored. Between the terms we that do depend on n, we can

choose between $0.001n^3$ and $0.025n$. So, 0.025 is the bigger than 0.001, so that does not matter in the long run. n^3 makes the term $0.001n^3$ *grow* a lot faster than $0.025n$. For example, if we fill in 10 for n, n^3 is 1000, and n is still 10, so $0.001n^3$ becomes 1 and $0.025n$ becomes 0.25. When we talk about complexity, we are always interested in ever larger input sizes – in *growth*. We will therefore always ignore the slower growing terms, and select the terms with the largest growth. This term is the *dominating term*. In this case the dominating term is therefore $0.001n^3$.

To get the big-Oh of an expression, we need to select the dominating term – $0.001n^3$ – and strip it of its forefactors. In this case, this leaves $O(n^3)$.

Special things that can happen in expressions is fractions in the constant c in n^c-terms. $n^{0.5}$ is the square root of n, i.e., $\sqrt{n}$. We note that this is still bigger than for example the logarithm of n, in terms of big-Oh notation. So $O(n^{0.5})$ grows faster than $O(logn)$. Can you now solve 1.?

2.4. Problems and their complexity

The complexity of problems is typically expressed in terms of the time complexity of the algorithms that solve them. As an example, Table 2.4 contains common tasks that are used in many applications and/or more complex algorithms.

Task	**Complexity**	**Algorithm**
Looking up a value in a lookup table (as in CPSE1)	$O(1)$ *(constant)*	array access
Calculating the maximum value in a sorted array	$O(1)$	array access
Checking whether an item is in a sorted array	$O(\log n)$	binary search (see Section 6)
Calculating the maximum value in an unsorted array	$O(n)$ (linear)	loop over the array (Algorithm 1)
Sorting a list	$O(n \log n)$	e.g., Merge sort (Section 7)

Table 2: The complexity of well-known problems through associated algorithms. For a plot of these different functions (n, $n \log n$, etc.) see Figure 2.

Note that not all complexities are simply exponents of n. Logarithms appear in complexity results a lot as well. We will explain later why this is the case.

Furthermore, note that for some tasks the exponents might not be whole. For example, the best complexity result for the multiplication of two $n \times n$ matrices – which is a key subroutine in graphics, computer vision, machine learning, etc. – is $O(n^{2.3728639})$ using an algorithm invented in 2014 by Le Gall [11]. In fact, matrix multiplication is such a common operation, that in other papers that use matrix multiplication as a subroutine,

the complexity of this multiplication is written as $O(n^\omega)$, where the omega stands in for the exponent for the latest possible complexity results for matrix multiplication. It has been proven that ω is bigger than 2, but how low it can actually go is still the subject of ongoing research.

2.4.1. Examples: Order! – answers

And hereby the answers:

	Expression	Dominant term(s)	O(...)
0.	$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
1.	$500n + 100n^{1.5} + 50n \cdot log(n)$	$100n^{1.5}$	$O(n^{1.5})$
2.	$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$	$2.5 \cdot n^{1.75}$	$O(n^{1.75})$
3.	$log(n) + n^2$	n^2	$O(n^2)$
4.	$nlog(n) + nlog(n)$	$nlog(n)$	$O(nlog(n))$
5.	$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
6.	$0.01n + 100n^2$	$100n^2$	$O(n^2)$
7.	$2n + n^{0.5} + 0.5n^{1.25}$	$0.5n^{1.25}$	$O(n^{1.25})$

3. Recursive Algorithms

In the previous section, we have discussed that were structured around nested for-loops. However, there are also algorithms that are structured quite differently. For example, let us take the classic example of (a naive algorithm for) computing the Fibonacci numbers (Algorithm 6).

Algorithm 6 recFibonacci

Input: a non-negative integer n
Output: the n-th Fibonacci number

1: **if** $n = 0$ **then**
2: **return** 0
3: **end if**
4: **if** $n \leq 2$ **then**
5: **return** 1
6: **end if**
7: **return** recFibonacci$(n-2)$ + recFibonacci$(n-1)$

This algorithm is *recursive*, i.e., it calls itself. There are no loops, but clearly its runtime increases as n increases, and rather steeply at that. So how do we analyse the time complexity of this algorithm?

First, we observe that within one call to the algorithm there are only simple computations (two comparisons and an addition; it is only the function calls that make it

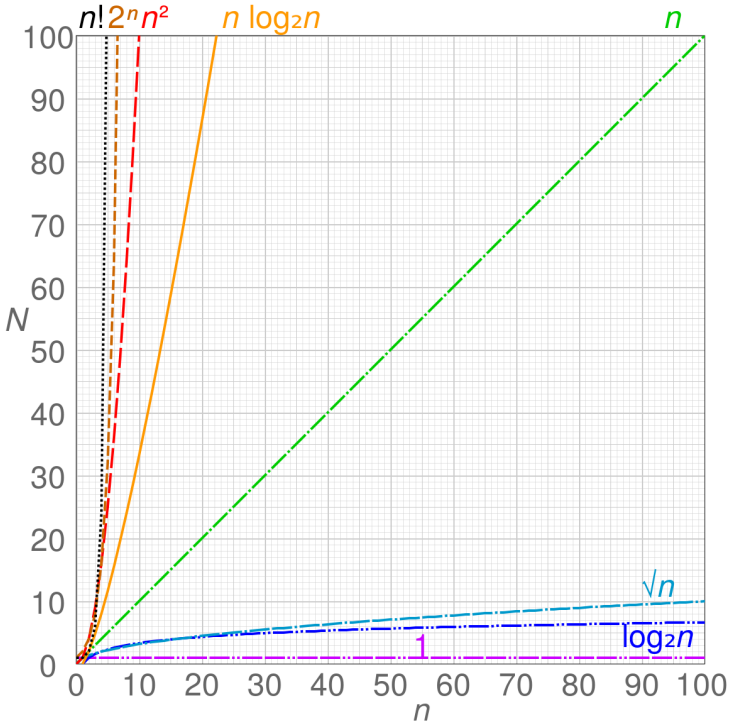

Figure 2: Common functions in complexity results as a function of input. Original figure by "Cmglee", taken from `https://commons.wikimedia.org/wiki/File:Comparison_computational_complexity.svg`.

expensive to execute. So, what we should count here is the number of function calls that Algorithm 6 does – recursively – as n increases.

Second, we observe that for each increase of n, the number of function calls roughly doubles. That is, when n is larger than 2, two new function calls are performed, specifically nrecFibonacci$(n-2)$ and nrecFibonacci$(n-1)$, which in turn (if $n-2$ and $n-1$ are still bigger than 2, will *both* execute two function calls. So the first call triggers 2 function calls, which will trigger 4 function calls, which will in turn trigger 8 function calls, and so on, until the base cases are reached. This leads us to the conclusion that this an instance of an exponential number of function calls with base 2. So the time complexity of Algorithm 6 is $O(2^n)$.

3.1. Exponential runtime

Exponential runtime, such as $O(2^n)$ is much worse than anything polynomial. See for example Figure 2, where 2^n (the dashed orange line) is plotted against other common complexity functions. Note there is only one function worse than exponential: the factorial function $(n!)$.

Exponential time complexity is worse than polynomial complexity. Even for really bad polynomials with high (but constant) exponents – for example $O(n^{42})$ –, the exponential

function 2^n will eventually overtake it and grow significantly more steeply. For example, 2^n overtakes n^{42} at 356 (rounded) already. Therefore, in Algorithmics it is critical to avoid algorithms with exponential time complexity whenever possible. In the next section, we discuss two tricks that may help to avoid such exponential time recursive algorithms, with a trick called "Memoïsation" (Section 4).

Of course, for some problems it is not possible – or at least highly unlikely – to get anything better than exponential time complexity. Furthermore, some problems are not solvable in finite time at all (not even exponential time). For more on this see Section 16.

3.2. A more precise runtime analysis*

This subsection is for the diehards, i.e., this will not be on the test, so do not worry if you do not get this, it is just a bit of additional information and fun.

After reading the explanation of the $O(2^n)$ complexity results, you might object along the following lines: *"Alorithm 6 does indeed make two function calls, but because in one of them n is decreased by* 2 *rather than by* 1*, the number of function calls does not quite double!"* and indeed you would be right. While the time complexity of naive recursive Fibonacci is exponential, the base is in fact a bit better than 2. If we want to find out how much better, we need to perform a few mathematical tricks. First, let us state the runtime of the algorithm as an equation:

$$runtimeFibo(n) \approx runtimeFibo(n-2) + runtimeFibo(n-1).$$

Note that we are ignoring constants (e.g., the number of computations executed inside a function call without the subsequent function calls) here, but they do not really matter anyway for complexity results. Now, for $runtimeFibo$ let us fill in an exponential function with an unknown base x,

$$x^n \approx x^{n-2} + x^{n-1},$$

divide by x^{n-2},

$$x^2 \approx 1 + x,$$

and then we can solve $x^2 \approx 1 + x$ as an equation (noting that we are only interested in positive solutions for this equation). Doing so – with the help of the ever faithful Wolfram Alpha on the internet[8] – yields:

$$x \approx \frac{1+\sqrt{5}}{2} \approx 1.62,$$

rounded upwards.[9] So the actual complexity of Algorithm 6 is approximately $O(1.62^n)$. This is still exponential – and therefore really bad indeed – but slightly better than $O(2^n)$.

[8] `https://www.wolframalpha.com/input/?i=x%5E2+%3D%3D+x+%2B+1`

[9] Coincidentally, $\frac{1+\sqrt{5}}{2}$ is a rather special number that crops up a lot in mathematics and nature in general, known as the "golden ratio".

3.3. Effective recursive algorithms and time complexity

The naive recursive Fibonacci example of Algorithm 6 makes recursive algorithms look bad. Especially as, as we will see in the next section, much more efficient algorithms are available. But why is it so bad?

The main reasons for Algorithm 6 being bad are the high *branching factor*, low *subproblem size* reduction, and high *recursion depth*. That is, a recursive algorithm typically reduces the problem at hand (in this case computing the n-th Fibonacci number) by calling the same function of a smaller instances of the problem called *subproblems* (i.e., computing the Fibonacci numbers for $n-2$ and $n-1$). Then it combines the results for these subproblems by doing some additional computation (in this case, adding them up). This is what is known as a *divide-and-conquer* strategy.

The steps of a recursive divide-and-conquer algorithm are:

1. **The base cases** – If the problem is smaller than a certain size (e.g., Fibonacci for $n \leq 2$), solve directly. Typically the problem is cut down to such a small size that it takes very little time to solve.

2. **Identifying subproblems** – If the problem is bigger than the base cases then hack up the problem into smaller subproblems (e.g., $n-2$ and $n-1$ for Algorithm 6). Of course these smaller problems should be useful, in the sense that it is easier to solve the full problem if the results for the subproblems are known.

3. **Recursion** – Calling the function on the subproblems. The number of subproblems for which the function is called is the *branching factor*.

4. **Recombination** – using the results for the subproblems to more quickly find a solution to the full problem.

The *recursion depth* is the maximum of the *recursion depths* for the subproblems plus 1, with the recursion depth of a base-case being 1 – which is of course (as it should be) a recursive definition. The recursion depth of Algorithm 6 for an input n is n: the recursion depth of its largest subproblem (size $n-1$), plus 1, i.e., $depth(n) = depth(n-1)+1$, so, $depth(n) = n$.

Effective recursive algorithms:

1. cut up the problem in much smaller subproblems (reducing the recursion depth), and

2. never compute a solution for the same subproblem twice (limiting the branching factor as much as possible).

Algorithm 6 fails on both accounts: the subproblems are too large, and the same subproblems are solved many times. In the next section, we will tackle problem 2, and show that this is in fact sufficient to make Algorithm 6 efficient.

For the diehards* The runtime $\mathcal{T}$ of a recursive algorithm, for an input size n, can be written as an equation:

$$\mathcal{T}(n) = \mathcal{T}_{\texttt{recombine}}(n) + \sum_{i \in \texttt{subproblems}} \mathcal{T}(n_i).$$

where $\mathcal{T}(n_i)$ is the time it takes to solve subproblem i assuming that the size for that subproblem is n_i, and $\mathcal{T}_{\texttt{recombine}}(n)$ is the time it takes to combine the results for the subproblems into the solution for the original problem of size n. For Fibonacci, we already worked out this equation in Section 3.2.

A great example of a recursive algorithm is `mergeSort` (which we will discuss in Section Section 7.4), which sorts an array of length n by recursively splitting the array in half (i.e., the subproblems are of size $\frac{n}{2}$, sorting both halfs, and then combining these two sorted lists in only n time. This makes the above equation $\mathcal{T}(n) = n + 2\mathcal{T}(\frac{n}{2})$, which when you solve it is just $O(n \log n)$ (just trust us on that one). If you want to know more about this splendid algorithm, stay tuned for next lecture and Section 7.4.

You will not be asked to solve such equations on the test – we will provide you with the results if needed, and when you need them in practice you can use equation solvers (such as Wolfram Alpha) to find the answers for you. It is however illustrative and useful to know where these complexity results come from.

4. Memoisation

One of the reasons why carelessly written recursive algorithms (and the resulting code) can get out of hand in terms of complexity (and resulting actual runtime) is that an answer for the same subproblem is computed recursively multiple times. For example, in the naive Fibonacci algorithm (Algorithm 6), if we call the function with $n = 6$, the function call with $n = 3$ happens three times ($5, 4, 3$ $4, 3,$ $5, 3$), and then recurses in the exact same way. This is unnecessary: the Fibonacci numbers do not change, so the function call with $n = 3$ will return the same result every time.

An elegant way of preventing this is "memoisation". This term comes from the Latin word *memorāre*: to remind or bring to mind. Basically the idea is: the algorithm will store the answers to the subproblems, and will immediately return the answer the second (or third or forth...) time the function is called with the same arguments. For Fibonacci the resulting algorithm is Algorithm 7.

Algorithm 7 mFibonacci

Input: a non-negative integer n,
an array `answers` of at least length $n+1$ (default value `null`)
Output: the n-th Fibonacci number

1: **if** `answers` $=$ `null` **then**
2: `answers` $\leftarrow$ a new array of length $n+1$ (at least length 2) filled with -1
3: `answers`$[0] \leftarrow 0$
4: `answers`$[1] \leftarrow 1$
5: **end if**
6: **if** `answers`$[n] = -1$ **then**
7: `answers`$[n] \leftarrow$ mFibonacci$(n-2,$ `answers`$)$ + mFibonacci$(n-1,$ `answers`$)$
8: **end if** ▷ *Make sure to pass* `answers` *by reference.*
9: ***return*** `answers`$[n]$

This algorithm is much more efficient than Algorithm 6; the recursion is only being done when the value is not known yet. Each value in the array is only computed once, and returned immediately thereafter. So, a function call to mFibonacci with a specific n might happen twice at most, but then the recursion stops. This means that the complexity is reduced from (about) $O(2^n)$ in Algorithm 6 to $O(n)$ in Algorithm 7. From exponential to linear complexity is about the biggest complexity gain possible in algorithmics, and all we needed to do is just remember something!

In general, to apply memoisation, an array is created with the necessary number of fields to hold all the necessary results for the subproblems. Such an array - plus associated functions to set and get the right elements of this array - is called a *lookup table* in the context of memoisation. We used this name in Table 2.4, where we indicated that looking up a value in a lookup table is $O(1)$, i.e., constant, which is as cheap as it can get.

Please note that – unlike in the Fibonacci case – the subproblems might not just be integer values between 0 and n. Therefore, some translation might be necessary to find the right indices in the array. As long as this translation from the arguments of the function to the right array indices is cheap (i.e., $O(1)$), this is of course not a problem.

Furthermore, please note that the array that holds the results for the subproblems might have to be 2 or 3 dimensional. You will see an example of such a problem in the practical assignments.

4.1. At Runtime Memoisation and Compile-time Memoisation

At this moment you might recall that during CSPE1, you created a *lookup table* already. Specifically, in the assignement with the clock, you created a lookup table to hold the answer to various calls to a function that (helped) determined the positions of the hands of your clock. Again, this is an instance of only ever computing a value for the same arguments of a function call once. In other words, you already applied memoisation before. Albeit for an entirely different use case. (Is it not awesome how the same idea

can help you with multiple problems?)

In CPSE1, the lookup table was created to not only ever compute the values in the lookup table once (though that is indeed welcome), but also – and in that case more importantly – to move the computation from runtime to compile time, so that the computation can be done when more computational resources are available (i.e., on your laptop rather than on the Arduino). This is a common feature of algorithmics for embedded software: you need to analyze the time complexity on the embedded system separately from the time complexity at compile time, or other parts of the system where more computational resources are available (e.g., imagine a robot that is connected to a computer via Wifi or Bluetooth that might perform intensive computation for it). Often, it will be worth it to move computation from the embedded platform to other hardware. In other words, keeping the time complexity of the algorithms that need to run on the microprocessor is key to making these systems run smoothly.

4.2. Memoisation is not free

"There ain't no such thing as a free lunch" as they say. This is a core principle in much of computer science – in optimisation there is even a formal "no free lunch theorem". Memoisation is often awesome, and can reduce the runtime of algorithms tremendously (as we have seen with Fibonacci). However, it does come at the price of having to use more memory.

This is worth reminding oneself of, as embedded systems typically not only have limited computational capacity, but also limited memory. In the case of the clock in the CPSE1 practical assignment, the extra memory used by the lookup table was actually less than the amount of memory that it would have taken to load the functions required instead onto the Arduino, so there all was well. But of course this is not always the case. When the lookup table has a lot of fields (and 2-, 3-, 4-, etc. dimensional lookup tables quite quickly have quite a lot of fields), it is typically more memory-intensive to put a lookup table in memory, than to put the functions in memory that are needed to compute its values.

The other way around is often also possible: reducing the amount of memory used by doing more computation. In fact, data compression (which, incidentally, you also did during CPSE1 with the LZ-compressor and -decompressor assignment), is a great example of this.

For an even more extreme – and admittedly less-than-serious example, see πfs (the π file system, `https://github.com/philipl/pifs`). πfs stores all its files by using sequences of digits of the number π. This is in fact possible: all finite sequences of digits (i.e., bytes) exist in the digits of π, so you just need to remember the right starting point as it were to retrieve a file – which might[10] lead to a lot of compression. Of course, there are no guarantees as to how long it will take you to find this starting point though... As the github page puts it: "Why is this thing so slow? It took me five minutes to store a 400 line text file!".

[10]Sadly, this is a big if though, as this starting digit contains as much information as the original file, the starting digits will on average also be very large.

4.3. Memory complexity

In algorithmics, it is often possible to trade of memory for computation, and vice versa. Both are key resources, so we need to be careful about both. Therefore, we are often not only concerned with *time complexity* but also with *memory complexity*.

We can analyse the memory complexity of an algorithm in much the same way as we analyse the time complexity of algorithms. For example, Algorithm 7, holds a lookup table with $n + 1$ values, i.e., $O(n)$. An n by n lookup table would be order $O(n^2)$, an n by m lookup table $O(mn)$, and so forth.

Again, of course, it is not quite that straight forward. Sometimes, we use Data Structures (see Section 5) that may be dynamic in size, so we would have to analyse it is largest possible size during execution. Again, we are interested in the worst-case scenario.

Furthermore, not all memory is created equal. One might for example argue that the naive recursive Fibonacci implementation (Algorithm 6) also has a memory complexity of $O(n)$. Specifically, function calls require putting things on the stack[11], specifically the value of the link register and all values kept in protected registers, so a function call typically requires $O(1)$ memory (unless e.g., arrays of order $O(n)$ or higher are copied with the function call). But recursion does a whole series of function calls, making the the stack grow. The worst case of the number of link registers (and other things) on the stack is exactly the *recursion depth* (that we defined Section 3.3), which for Algorithm 6 (and Algorithm 7 for that matter) is also n.

Finally, please keep in mind that different types of memory exist. Typically, compilers reserve a limited amount of memory for the stack, aside from the rest of the memory reserved for the program. This stack size is typically much smaller than what is reserved for the rest of the memory. It is therefore possible that a recursive algorithm might give a stack-overflow error, while an equivalent non-recursive algorithm with the same memory complexity may not give an out-of-memory error. It is usually possible to reserve a larger amount of memory for the stack via compiler arguments.

5. Data Structures

In the previous sections, we have discussed time and memory complexity, and seen that algorithms have a profound influence over both. However, algorithms specify the structure of the code, while this is not all that determines the runtime and the memory usage; the other key component of keeping runtime and memory usage in check is how to store data. This section is about exactly that: data structures.

5.1. Containers

A highly prevalent type of data structure is *containers*. Containers are objects whose sole purpose is to store and retrieve other objects efficiently. As such, containers are often templated in programming languages. For example, an array *of integers*, a vector *of floats*, a list *of students*, etc.

[11] See Section 5.5 for a further explanation of stacks as a data structure.

Containers may implement different methods, depending on their intended usage, but ultimately, they all have some basic methods/features:

- A way to *store* objects in the container
- A way to *access* objects in the container
- A way to *traverse* the objects in the container (i.e., systematically go over each individual object) in the container.

Object traversal is often referred to as *iterating* over the objects. As such, containers often implement an *iterable* interface in programming languages. The container then is able to return an *iterator* (possibly itself) which can be initialised, and implements a function to retrieve the "next" object.

This of course sounds very abstract, so let us look at different instances of, and types of containers.

5.2. Arrays

The most common container data structure is probably the array. The array stores objects on a given index ($a[i] = x$), and access is also done using this index ($x = a[i]$). Traversing an array is also straightforward: one can loop over all the indexes in the array, and because an array has a fixed size (e.g., N), the indices are always from 0 to this size minus one ($N - 1$).

Arrays are simple, easy to translate to assembler code, and implemented in almost every modern programming language. Of course, arrays also have downsides: they are fixed in size, the elements of the array need to be initialised somehow, and locating and replacing objects in the array may be less than efficient.

That being said, arrays often form the basis of more extensive data structures that have more capabilities and/or additional properties (desirably for their intended use). Note though, that unless special tricks are applied – see Section 8.1 – data structures implemented using arrays will have a fixed capacity.

5.3. Sets

An example of a container class that you have already implemented (in CPSE1) is the Set. A set is a container with a special requirement: it may not contain duplicates. Therefore, when adding an element to the set, it needs to checked whether the set already contains the given object. A result of this, is that a set requires objects that can test for equivalence (e.g., via the `==` operator).

When the set has a fixed (max) capacity, it is possible to implement a set on top of an array. Specifically, a set class would need:

- an array of length `maxCapacity`,
- a counter with how many elements are currently in the set, and

- an `insert` method that respects the set property (i.e., it needs to check whether an equivalent object is already in the set, and not insert the item if it is already in the set)
- in addition to methods to *remove* and *traverse* objects in the set, but those are relatively standard.

These methods are all little algorithms in their own right. It is key to keep the time complexity of these methods in check, as containers are typically used frequently by the programs. Furthermore, it is often the case that implementing the data structure in one way will lead to efficient insertion, may lead to slower removal or more memory usage. The best way to implement a container may thus heavily depend on which methods are being called most frequently (e.g., inside of another algorithm).

So, let us have a look at possible algorithms for *insert* (Algorithm 8) and remove (Algorithm 9 for the above-mentioned array-based implementation of a Set.

Algorithm 8 set.insert(x)

Object variables: an unsorted array `elems`, an integer `size`
Input: a new object to insert x
Output: boolean indicating whether x was successfully inserted

```
1:  if size=length(elems) then
2:      return false                ▷ max cap.
3:  end if
4:  for i ∈ 0...size−1 do
5:      if elems[i] = x then
6:          return false
7:      end if                      ▷ already in set
8:  end for
9:  elems[size] = x                 ▷ add to set
10: size++
11: return true
```

Algorithm 9 set.remove(x)

Object variables: an unsorted array `elems`, an integer `size`

Input: an object to remove x

Output: boolean indicating whether x was successfully removed

```
1: for i ∈ 0...size−1 do
2:     if elems[i] = x then
3:         elems[i]=elems[size–1]
4:         size−−
5:         return true
6:     end if
7: end for
8: return false
```

For both *insert* and *remove* the time complexity is $O(\texttt{size})$, i.e., the number of elements currently in the set.`size` is in turn limited by the maximum capacity of the array. For remove, this would also be the case if it were not a set, as in order to remove an object from a collection implemented with an unsorted array, all elements that are currently in that collection could have to be compared against x to test whether x is in that collection. For *insert* however, the fact that it takes $O(\texttt{size})$ time is caused by the fact that it is a set, i.e., because of the condition that the set may not contain doubles, it must be checked whether x is already in the set, which takes $O(\texttt{size})$ time. Adding x without checking whether x is already in current elements would only take $O(1)$ in a

collection implemented using an unsorted array. So, the set property makes insertion significantly more difficult.

So, could inserting into a set be made cheaper? For example, what if the array of elements were sorted? In that case, finding the element in the sorted array would be cheaper (as we will explain in Section 7). However, inserting an element into a sorted array could still take $O(\texttt{size})$, i.e., when the new object is the smallest, it should be inserted at index 0, which means that all the elements currently in the array would have to move one place to the right (that is, to their current index+1). As such sorting the array does help to make insertion cheaper. So, is there another way? Indeed there is, but not using an array. In the next lecture, we will have a look at a completely different data structure (i.e., trees) that could be used to make inserting and removing objects cheaper.

5.4. Array lists

Now, let us compare to a container data structure that does not require the set property, but is implemented using an array as its basis. This is called an ArrayList[12].

When such an ArrayList has a fixed (max) capacity, it is possible to implement a set on top of an array. Specifically, a set class would need:

- an array of length `maxCapacity`,
- a counter with how many elements are currently in the ArrayList, and
- an `insert` method, as well as a *remove* method.

The remove method is identical to that of set (Algorithm 9, in Section 5.3). However, the insert method becomes much simpler. Because no check needs to happen whether x is already in the set, inserting into an ArrayList takes only $O(1)$ (constant) time.

So, the ArrayList can have a variable number of objects contained. Note however, that because the array is always `maxCapacity` long, the memory requirements are always the same.

We further note that it is possible to get around the max capacity by a trick. That is, when trying to insert a new item to a "full" ArrayList, we could simply create a new array to contain the elements that is much longer, and copy all elements from the old array to the new array. Note however, that this takes $O(n)$ time, so this would every now and then seriously impact the runtime of insertion into an array list. Nonetheless, standard implementations of ArrayLists in various programming languages typically do this. We will explain why this is so in Section 8.1.2.

5.5. Stacks

A container type data structure that can be implemented highly efficiently using an array (if having a maximum capacity is okay), is a Stack. Stacks are highly prevalent in the literature, and are indeed one of two data structures typically implemented on chips.

[12]Though, abusively, the name `vector` is used for this in C++.

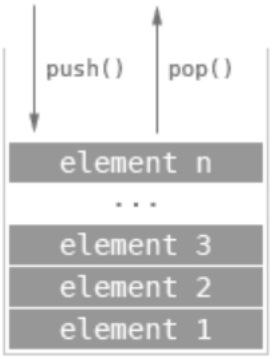

Figure 3: A graphical depiction of a stack, taken from `https://commons.wikimedia.org/wiki/File:Stack_data_structure.gif` (created by Hyperthermia).

A stack is a container a simple requirement:

- The insert and remove methods can only change the "top" op the stack, according to the last-in-first-out (LIFO) principle.

This explains the metaphorical name "stack". On a stack of objects (e.g., plates), only the top can be safely removed, and an object can only safely be added to the top. In a stack, the insert method is called *push*. The remove method is called *pop*. An illustration of a stack is given in Figure 3

Algorithm 10 stack.push(x)

Object variables: an array `elems`, an integer `n` (current number of elements in the Stack)
Input: a new object to insert x
Output: boolean indicating whether x was successfully pushed

1: **if** $n=$`length(elems)` **then**
2: **return** `false` ▷ max cap.
3: **end if**
4: `elems`$[n] = x$ ▷ add to stack
5: $n{+}{+}$
6: **return** `true`

Algorithm 11 stack.pop()

Object variables: an array `elems`, an integer n (current number of elements)
Output: the last object added to the stack

1: **if** $n = 0$ **then**
2: **return** `null`
3: **end if**
4: $n{-}{-}$
5: **return** `elems`$[n]$

Due to these simple requirements both inserting (pushing, Algorithm 10) and removing (popping, Algorithm 11) objects on/from stacks is order $O(1)$ time.

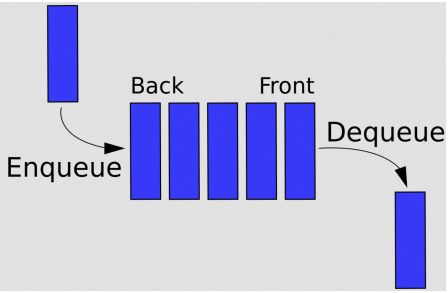

Figure 4: A graphical representation of the Queue data structure and its methods. This figure was taken from `https://commons.wikimedia.org/wiki/File:Data_Queue.svg`, and was created by User:Vegpuff.

5.6. Queues

A different data structure with slightly more complex requirements is the *queue*. Like a queue at the supermarket, the requirements are:

- The insert (queue) method can only change the "back" of the queue, according to the first-in-first-out (FIFO) principle.
- The remove (dequeue) method can only change the "front" of the queue, according to the first-in-first-out (FIFO) principle.

A graphical depiction of a queue is provided in Figure 4.

Because of the slightly more complex requirements, it is slightly more complex requirements the data structure must now keep track of where the front and the back of the queue is. However, if the queue is implemented via an array, it is still possible to have both insertion (queuing) and removal (dequeuing) in $O(1)$ time. This is possible because the queue can be implemented in a circular manner, so that the elements in the array still do not need to be shifted when elements are added or removed.

Algorithm 12 queue.queue(x)

Object variables: an array `elems`, three integers n (the current number of elements), and `front` and `back`
Input: a new object to insert x
Output: boolean indicating whether x was successfully queued

```
1: if n=length(elems) then
2:     return false                ▷ max cap.
3: end if
4: elems[back] = x                 ▷ add to queue
5: back++
6: if back = length(elems) then
7:     back ← 0
8: end if
9: n++
10: return true
```

Algorithm 13 queue.dequeue()

Object variables: an array `elems`, three integers n (the current number of elements), and `front` and `back`
Output: the oldest object added to the queue

```
1: if n = 0 then
2:     return null
3: end if
4: result ← elems[front]
5: front++
6: if front = length(elems) then
7:     front ← 0
8: end if
9: n−−
10: return result
```

Please note that, in the algorithms the front is the leftmost element, rather than the rightmost, contrary to Figure 4.

5.7. Dictionaries

Note that the topic of dictionaries is not discussed during the lectures, but will be on the test. Therefore, please read this section carefully, and ask questions to your "klassedocent" if you have any.

Another common datatype is the dictionary. A dictionary is a container in which objects, called `values`, in the container are identified using a `key`. For example, we could have a dictionary in which student objects are identified via a student number (an int) as a key. Dictionaries store (`key`, `value`)-tuples.

In order for keys to correctly identify the accompanying values, it is required that each key is unique. Dictionaries are thus highly similar to sets (Section 5.3), with the difference that the objects are stories in a (`key`, `value`)-tuple, and that it is the `keys` that are required to be unique, rather than the objects themselves (i.e., `values` need not be unique).

Because of the similarity between sets and dictionaries, each algorithm (insertion, removal, search, and traversal), can be implemented in the same way for a dictionary as for a set, with the same time complexity.

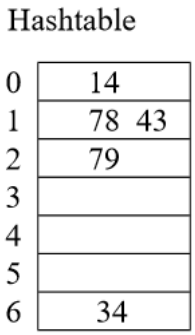

Figure 5: A example of a hashtable with a collision at index 1. Figure by Joop Kaldeway.

5.8. Hashmaps

Note that the topic of HashMaps is not discussed during the lectures, but will be on the test. Therefore, please read this section carefully, and ask questions to your "klassedocent" if you have any.

A common way to trade off memory for computation, at least in expectation, is HashMaps. A HashMap is a specific case of a lookup table (see Section 4) where the index in the table of an object is its *hash*. This hash value is determined by a hash function, which does nothing other than to compute some (rather unstructured) value to function as the index (or key if you will) for insertion and search in the array.

A HashMap consists of the following components:

- A hash table (typically an array)
- A hash function (to determine the indices of objects in the table), and
- A collision strategy, i.e., what to do if – as misfortune might have it – two objects have the same hash.

For example, let us take a HashMap based on an table (i.e., an array) of length 7, and let us use $f(x) = x \mod 7$ as the hash function (mod being the modulo function, denoted % in many programming languages). If we now insert the values 14, 78, 34, 79, and 43, be would obtain the hastable in Figure 5. In this figure we see that at index 1, there are now two elements. This is called a *collision*. Of course, the reason for this collision occurring is that we have made the hash table much too small. In order for collisions to occur as little as possible, we want to make HashMaps that are sparsely populated, i.e., with much more space than actual objects in the table. However, it is never possible to completely rule out collisions. Therefore, we need a collision strategy.

For example, in a *linear hashing collision strategy*, we would simply try to insert the new value in the next index, and if that is full, in the subsequent index, and so on. However, this may lead to larger clusters of filled fields. This is undesirable, as it makes searching more expensive. That is, when we need to search for an object later, we need

to search from the has value until the next empty spot, if we want to be certain that the object we are searching for is not in it. Slightly more sophisticated collision strategies therefore take bigger jumps in indices when collisions occur.

The collision strategy for example used by the JAVA programming language is to have another container (e.g., a list as we will define in Section 8.1) at each index, rather than inserting the objects directly into the hash table. Of course, these containers will take more time to search (see Section 6). But, as these containers will contain very few elements, this search time will typically be negligible in practice (as long as the HashMap is sufficiently sparsely populated).

6. Search

A key algorithm connected to container data structures that we have not yet discussed is search, i.e., *"How do find an object in a container"?* Sometimes, this is just to check whether we already have the object in the container, and sometimes, it is to retrieve the correct object (e.g., the correct value for a given `key` in a dictionary).

Searching for objects is a highly common operation, and often the most intensively used method for container-type data structures. We also note that any remove method needs to search for the object in the container as well.

If we have an unsorted array as the basis for storing our objects, we have already seen that finding an object takes order $O(n)$ time. That is, we have to loop over the entire array in order to find the object in our container.

However, it can help if we impose the constraint that the objects in the array are stored in a sorted order. So, we will make sure that element 0 is smaller than element 1, element 1 smaller than 2, and so on.[13]

6.1. Logarithmic time complexity

In this subsection we assume that you know what a logarithm is. We will be using logarithms with base 2. If you are unfamiliar with logarithms, please check out one of the many internet tutorials on the matter. For example the one by Khan academy[14].

Counting the number of nested for-loops is sufficient for determining the time complexity of many algorithms. However, for some tasks it is possible to be faster than polynomial. If we get a *sorted* array (e.g., of integers) as input, it is possible to test whether a given value is in that array in less than $O(n)$ time – $O(n)$ being the complexity of just looping through the entire array.

[13] Of course, this means that we have to be able to sort the elements in our array. We will discuss sorting algorithms in Section 7.

[14] `https://www.khanacademy.org/math/algebra2/x2ec2f6f830c9fb89:logs/x2ec2f6f830c9fb89:log-intro/a/intro-to-logarithms`

Algorithm 14 binarySearch

Input: a sorted array: a, and an item to find: x
Output: either the index of x in a

```
1: l ← 0
2: r ← length(a) − 1
3: while l ≤ r do
4:     m = l + (r - l) / 2;
5:     if a[m] = x then
6:         return m
7:     end if
8:     if a[m] < x then          ▷ If x is bigger than a[m], ignore everything left of m
9:         l ← m + 1
10:    else                      ▷ If x is smaller than a[m], ignore everything right of m
11:        r ← m − 1
12:    end if
13: end while                    ▷ If the algorithm gets here, x was not in a
14: return not found
```

Let us have a look at Algorithm 14, which is commonly known as binary search. At each iteration of its while-loop, it is able to cut the "search window" (the part of the array that may contain x) in half by inspecting the middle m of its current search window, i.e., everything between the indices l (left) and r (right). If the middle is smaller than x, x and only be to the right of index m. This comes from the fact that the array is sorted. Conversely, if m is larger than x, x can only be to the left of x in the array.

The number of iterations required to check whether x is in a sorted array (and return its index if found) is the number of times it takes to reduce the search window to 1 remaining element. For example, if the number of elements would be 16, iteration 1 reduces the search window to 8, iteration 2 to 4 elements, iteration 3 to two elements, and iteration 3 to one element, after which, in iteration 4, we need to check this final element. As you can see, we can check double the amount of elements for every extra iteration i, 2^i. E.g., for 4 iterations we can have $2^4 = 16$ elements. The reverse of this is the ${}^2\log$ (which we will write as simply log from now on). Therefore, Algorithm 14 has a time complexity of just $O(\log n)$.

7. Sorting

As we have seen in the previous section, finding items in a sorted array is much faster than in an unsorted array. When a lot of searches need to happen, it may thus be well worth it to keep the elements in a collection in a sorted array. Furthermore, for many other algorithms that take an unsorted set of objects as input, it helps to sort the objects beforehand.

In this section, we will first discuss how expensive it is to insert a new object into an already sorted array. Then we will proceed to various sorting algorithms that take an

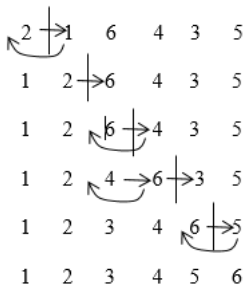

Figure 6: An example of the execution of insertion sort (made by Joop Kaldeway). At every timestep the current index (verticale bar) moves 1 place to the right, and the elements to the left of this index become sorted.

unsorted array as input, and produce a sorted array containing the same elements.

7.1. Inserting one element into a sorted array

Inserting a new object into an already sorted array takes $O(n)$ time. This is not because it is expensive to find the right place to insert the new object; we can use an algorithm highly similar to binary search (Algorithm 14) to return the right location. Specifically, at the end of the algorithm, after the while loop, the algorithm should check whether $a[m]$ is bigger than x, and if so x should be inserted at m, and else at $m + 1$. It is the insertion given the index which takes $O(n)$ time. This is because all elements at and after this index should be moved to the right (to their current index plus one).

Exercise: try writing the pseudo-code for the algorithm described in the text of this subsection, and show this to your "klassendocent".

7.2. Insertion sort

Now that we know that we can insert a new object into a sorted array in order $O(n)$, this begs the question whether we can use this as the basis for an efficient sorting algorithm. The basic idea of this algorithm – known in the literature as *insertion sort* – is shown in Figure 6. In each iteration of the algorithm the part of the array that is sorted increased by one element. I.e., at each iteration, we "insert" the element at index i of the sorted part of the array: elements 0 to $i - 1$.

Algorithm 15 insertion sort

Input: an unsorted array: a
Output: a, but now sorted

1: **for** $i \in 1...length(a) - 1$ **do**
2: $x \leftarrow a[i]$
3: $j \leftarrow$ find the place to insert x between index 0 and $i - 1$ ▷ $O(\log i)$
4: Move all elements of a between j and $i - 1$ one index to the right ▷ $O(i)$
5: $a[j] \leftarrow x$
6: **end for**
7: **return** a

The pseudocode for insertion sort is given in Algorithm 15. Please note that in this algorithm we describe what to do in English, rather than in code-like statements. This is to make the algorithm better understandable to the reader, without going too much into detail. This is possible because we know that the reader, i.e., you, would know how to implement these steps efficiently.

For insertion sort the worst-case input would be an array in reverse order. In that case, at each iteration, it would have to move exactly i elements, as at each iteration the index to insert the new element would be 0. If the length of the array is n, then it has to more $\frac{n}{2}$ per iteration of the outer loop on average. Because this outer loop is executed $n - 1$ times, this leads to a time complexity of $O(n^2)$. Surely, we should be able to do better.

7.3. Quick sort

We have seen that the most straight-forward way of sorting, insertion sort (Algorithm 15), has a time complexity of $O(n^2)$. Now let us see whether we can do better.

Maybe we can use a similar trick to the one we have used to find an object in a sorted array. So, we would try to start somewhere in the middle, and divide up the elements in everything smaller and larger, having to sort two halves of the array separately – leading to a recursive scheme. This is the central idea behind an algorithm that is known as *Quick Sort*. The element we pick to divide the array up into two (everything smaller to the left, and everything larger to the right), is called the *pivot*. The operation of splitting the array in two parts (smaller and bigger than the pivot) is called *partitioning*. A graphical depiction of this idea is given in Figure 7.

There are now two problems in making this idea work:

1. how to implement the partitioning operation efficiently, and
2. how to pick the pivot such that the array is split in (roughly) equal parts.

For the first problem there is an elegant solution: we can use the swapping of elements in the array. Please take a moment to appreciate (and understand) how this works looking at Algorithm 17, using the partitioning scheme by Lomuto. Using this partioning scheme, QuickSort (Algorithm 16) becomes a straightforward recursive algorithm.

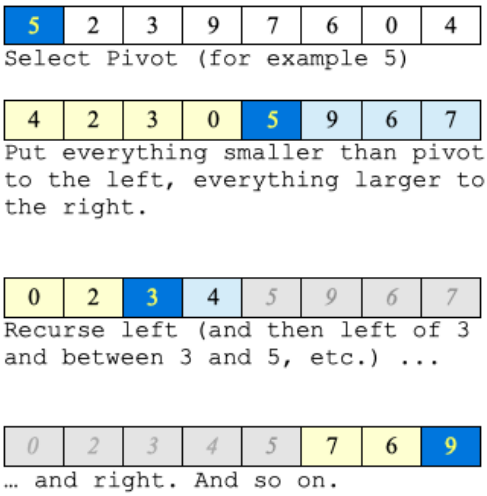

Figure 7: The central idea behind the quick sort algorithm (note, this example does not use a particular implementation, it just depicts the central idea).

Algorithm 16 quicksort

Input: an unsorted array a, two indices, lo and hi
Output: a, which is now sorted between hi and lo

1: **if** $lo \geq hi$ **then**
2: **return** a ▷ base case
3: **end if**
4: $p \leftarrow \texttt{partition}(a, lo, hi)$
5: $a \leftarrow \texttt{quickSort}(a, lo, p-1)$
6: $a \leftarrow \texttt{quickSort}(a, p+1, hi)$
7: **return** a

Algorithm 17 partition

Input: an unsorted array a, two indices, lo and hi
Output: i, an index for where everything left of i is smaller in value, and everything right larger.

1: `pivot` $\leftarrow a[hi]$
2: $i \leftarrow lo$
3: **for** $j \in lo \ldots hi-1$ **do**
4: **if** $a[j] \leq$ `pivot` **then**
5: swap values of $a[i]$ & $a[j]$
6: $i{+}{+}$
7: **end if**
8: **end for**
9: Swap values of $a[i]$ and $a[hi]$ ▷ ... where the pivot is.
10: **return** i

The second problem however, we cannot actually solve effectively. That is because the best possible pivot would of course be the one that splits the elements in two equal parts. If this is successful we would only need to do $O(\log n)$ splits, making the entire algorithm $O(n \log n)$. To find this best pivot however, we would need to know what the middle element of the elements to sort is. But in order to do this, we would have to sort

these elements, which defeats the purpose. On average, if we just select a random pivot from the remaiming elements, we will probably be fine. If for some reason however, we always pick either the largest or the smallest element, then we would have to recurse $O(n)$ times, and we would still end up with a $O(n^2)$ time complexity.

As the worst-case complexity is $O(n^2)$, our quest for a sorting algorithm with better guarantees continues. It is worth noting though that due its elegance, good average performance, and little additional memory requirements, it is still often used in practice.

7.4. Merge sort

The problem with quick sort is that there is no way to guarantee that we always choose a good pivot. Therefore, the question becomes, can we split the sorting problem up into subproblems, in a way that does not depend on choosing such a pivot?

Indeed, there is. As the basis for this strategy, we make use of the following observation:

- If we have two sorted arrays, of respectively length l and m as input,
- then merging them into one sorted array of length $l+m$ only takes $O(l+m)$ time.

This is because we can start at index 0 for both arrays, and keep inserting the next element from the array which happens to be smallest. This operation we call `merge`, shown in Algorithm 18.

Algorithm 18 merge

Input: two sorted arrays, a and b
Output: a sorted array containing all elements from a and b.

```
1:  i, j, k ← 0, 0, 0
2:  res ← a new array of length length(a) + length(b)
3:  while i < length(a) ∧ j < length(b) do
4:      if a[i] < b[j] then
5:          res[k] ← a[i]
6:          i++
7:      else
8:          res[k] ← b[j]
9:          j++
10:     end if
11:     k++
12: end while
13: if i < length(a) then
14:     insert the remainder of a to res          ▷ Using a loop from i to length(a)−1
15: else
16:     insert the remainder of b to res
17: end if
18: return res
```

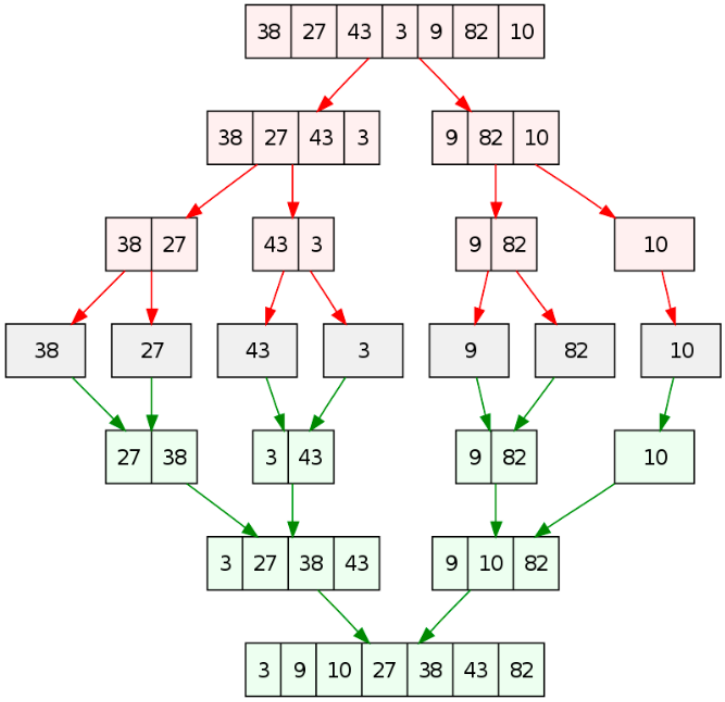

Figure 8: The idea behind the merge sort algorithm. (Image taken from `https://commons.wikimedia.org/wiki/File:Merge_sort_algorithm_diagram.svg`, own work by VineetKumar).

Using this merge operation to combine the results of the subproblems, we can simple split our array down the (index-based) middle, sort both halves, and merge the result. An example of this idea is depicted in Figure 8.

The resulting algorithm is called *merge sort*. Because we can now guarantee that the subproblems are indeed half the size of the original problem (i.e., the entire array that we have to sort), the recursion depth will just be $\log n$. At each recursion though, we have to first put all n elements into smaller arrays, and then merge them together again. This copying and merging takes order $O(n)$ time, making the total complexity $O(n \log n)$.

Algorithm 19 merge sort

Input: an unsorted array a
Output: a, now a sorted array.

1: **if** $length(a) = 1$ **then**
2: **return** a
3: **end if**
4: $m \leftarrow length(a)/2$ ▷ Cast to integer value
5: $l \leftarrow$ an array containing all elements of a between index 0 and $m - 1$
6: $r \leftarrow$ an array containing all elements of a between index m and $length(a) - 1$
7: $l \leftarrow$ `mergeSort`(l) ▷ This algorithm!
8: $r \leftarrow$ `mergeSort`(r) ▷ This algorithm!
9: $a \leftarrow$ `merge`(l, r) ▷ Algorithm 18
10: **return** a

8. Recursive Data Structures

In the previous sections of this reader – as well as the lectures – we have discussed recursive algorithms, and have seen that this can drastically reduce the time complexity for many common computational tasks. For example, linearly looping over an array in order to find an item is $O(n)$, but performing binary search (Algorithm 14) on a sorted array only takes order $O(\log n)$[15]. Furthermore, to sort an array, the straightforward approach using two loops, i.e., insertion sort (Algorithm 15) takes $O(n^2)$ time, but using a clever recursive trick we can get to merge sort, which takes only $O(n \log n)$ time.

This of course begs the question, if we can do recursion in algorithms, can we not also use recursion in data structures? That is, if we define data structures in terms of themselves, does that work, and if so what advantages can that bring?

8.1. (Linked) Lists

A well-known recursive data structure is the list. The list is an alternative to an array, i.e., a linear way to store data. Roughly speaking a list consists of:

- A head: an element of the list (e.g., an integer), and
- a tail: a list (i.e., the rest of the list).

If the tail is empty (or `null`) the end of the list is reached.

For example, Figure 9 depicts a small list of integers. The list is stored as objects called *nodes*. One node contains an element, and a pointer to the rest of the list, i.e., the next node.

The main advantage of this data structure over the array is that it has an "infinite" capacity, and is flexible in size. Another advantage is that while iterating over the nodes,

[15] Please remember that in this reader, we mean ${}^2\log n$ (the logarithm base-2) when writing $\log n$, as is common in the computer science literature.

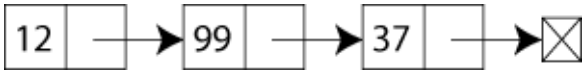

Figure 9: An example list of integers. The first "node" has a head: the integer 12, and a tail, a pointer to the next node in the list, i.e., the list starting from 99. Figure taken from `https://commons.wikimedia.org/wiki/File:Singly_linked_list.png`, own work by Derrick Coetzee.

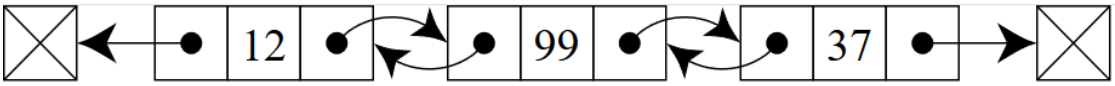

Figure 10: An example doubly linked list of integers. The first "node" has a head: the integer 12, and a tail, a pointer to the next node in the list, i.e., the list starting from 99, as well as a pointer to the previous node in the list, which is `null` in the case of the first node. Figure taken from `https://commons.wikimedia.org/wiki/File:Doubly-linked-list.svg`, own work by Lasindi.

it is simple to take one node out, i.e., if we want to remove the node with element 99 from the list, we need to set the pointer to the next node of the node with element 12 to the node with element 37 (and of course destruct the node with element 99).

The main disadvantage of lists with respect to arrays is that it is not indexed. This means that we cannot directly access the third node in the list, but we have to start at the beginning of the list, and follow the pointers to the next node, and the next node again. This means that accessing the x-th node in the list as a time complexity of $O(x)$.

While the list is uncommon in most imperative programming languages (like C++, JAVA and Python) – for reasons we will explain in Section 8.1.2 – it is one of the cornerstone data structures in many functional programming languages, such as Haskell. If you take functional programming as an elective subject later, you can therefore expect to see a lot more of them.

8.1.1. Doubly Linked Lists

Another disadvantage of the singly linked lists of Figure 9, is that is easy to iterate through a list in a forwards way, but it is difficult to go back. Therefore, aside from singly-linked lists, there are also doubly linked lists, in which each node has a pointer to the previous as well as the next node. See Figure 10 for an example.

8.1.2. A key side-note to linked lists

On most PC/laptop processors, using arrays is significantly faster than using linked lists. This is because in a linked list, the memory addresses of the nodes are unstructured and scattered around the (heap) memory. Therefore, object traversal in a linked list is slower, as the processor is accessing relatively random memory locations. In contrast, a contiguous memory structure such as a C-style array has a continuous memory block.

Most PC and other "fancy" CPUs will automatically cache contiguous chunks of memory while executing programs. And, as the access time for this cache is orders of magnitude lower than that of RAM, iterating over an array much faster in practice than iterating over a linked list. When using linked lists, this hardware-based advantage will not be available. Therefore, even when using unlimited capacity data structures, the standard implementation in many programming languages (such as C++) is typically still array-based.

When implementing a Stack or Queue with an unlimited capacity via an array, the trick is to just create a new array with double the capacity whenever an element is added to a currently full array. Of course this requires copying all the elements from the previous array to the new array costing $O(n)$ time. In practice, on PCs and other "fancy" CPUs however, the average runtime gain of caching is worth this expense. On microprocessors on the other hand, the caching advantage is typically not there. Therefore, on microprocessors is can still be worth it to use Linked Lists to implement container data structures that have unlimited (or at least uncertain max) capacity.

8.2. Trees

Another highly common data structure is the Tree[16]. A tree has a *root* node (i.e., the starting point of the tree). Each node has child nodes. When each node has at most two children it is a *binary tree*. Each child is itself the root of a *subtree*. A node that has no more children is called a *leaf* node.

Binary trees are highly common, but it is of course possible for each note to have more than two child nodes.[17] For example, Figure 11 (left) shows an example tree where the root node (the green 2 in the red circle) has two child nodes. The subtree rooted by the left child node (7) has a total of 6 nodes. The left child node itself has three child nodes (2, 10, and 6), two of which are leaf nodes (2 and 10). In this tree each node contains an integer, in addition to its list of child nodes.

Another example of a tree, this time to index words, is the prefix tree in Figure 11 (right). In this tree, going down the tree means adding to the prefix of a word. So the word "tea" can be found in a leaf node that descends from "te", "t", and "" (the empty root node). Such prefix trees are for example very useful for quickly finding dictionary entries.

Trees are very useful data structures, whose content can be adapted for usage in all kinds of algorithms. In this reader we will scratch the surface in two areas: trees as efficient containers optimised for search as well as well-performing for inserting and removing (Section 8.3), which are called search-trees. And trees that are created as part of so-called *tree search algorithms* for optimisation (Sections 11 and 12).

[16] Want bomen zijn relaxt.

[17] We could also say that linked lists are a special case of trees: a tree where each node has exactly one child node.

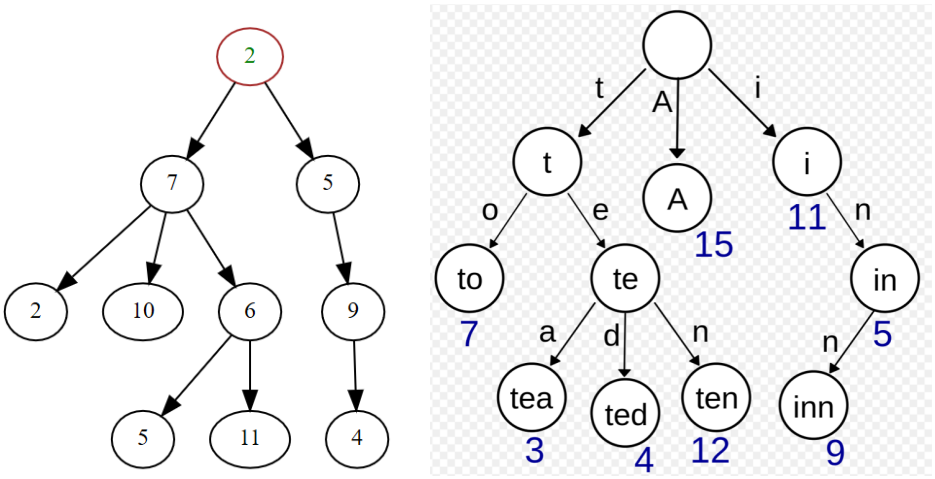

Figure 11: (left) an example of a tree of integers (right) an example prefix tree.

8.3. Search Trees

In Section 6, in Algorithm 14 we have seen that having an array sorted can reduce the time complexity of search to $O(\log n)$. This is because we can always ignore half of the array after a comparison. However, inserting into a sorted array is $O(n)$, as is removing an object from a sorted array. This is because, to keep the array sorted, we might have to shift (almost) all values in the array to the left (i.e., one index lower) or right (i.e., one index higher). This is unavoidable when using arrays.

So, what if we change the data structure from an array to something more resembling the binary search scheme, for which inserting and removing becomes cheaper. This is called a *binary search tree (BST)*. A BST has the following conditions:

- If n is a node in a BST, with an element of value v, then,
- all left descendants of n (i.e., its left child node, both its child nodes, their child nodes and so on...), n_d have values v_d that are smaller than v, expressed more mathematically[18], $\forall(d \in \texttt{left\_desc(n)}) : v_d < v$, and,
- all right descendants of n have values larger than v, i.e., $\forall(d \in \texttt{right\_desc(n)}) : v_d > v$.

If double values are allowed, then equal values should be place on either the left or the right *consequently*. An example of a BST can be found in Figure 12 (left).

Searching in a BST follows the same scheme as binary search for arrays. However, rather than a search window (two indices) in the array, now a current node in the tree is maintained; the window being it and all its descendants. For example, if we want to look for 5 in Figure 12 (left), we start at the root: 7. $5 < 7$ so we go to its left child,

[18] the $\forall$-sign means "for all". So the formula translates to: for all nodes that are a left descendant of n have a value lower than the value of n.

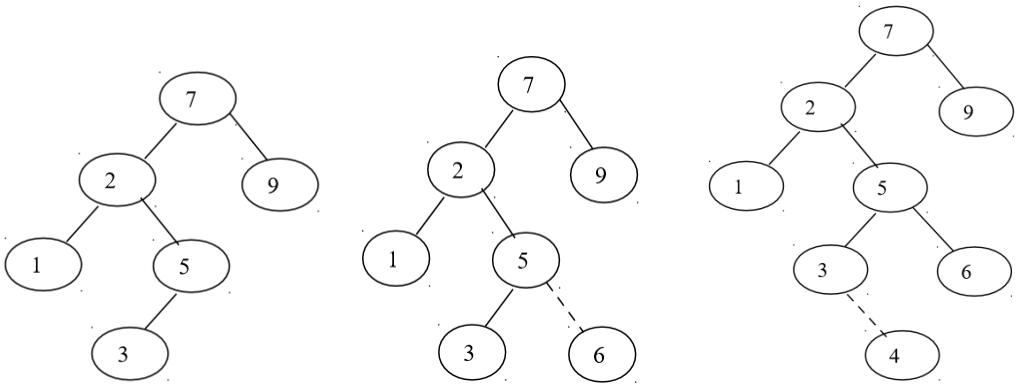

Figure 12: An example of a binary search tree: (left) original, (middle) after inserting 6, and (right) after inserting 4. Figures by Joop Kaldeway.

2. $5 > 2$, so we go to its right child, which brings us to 5, and we can return the node containing 5 (or just `true` depending on what we need).

Inserting in BSTs is straightforward too. We follow the tree down the same way as in search, and insert a new node as soon as we find an empty spot. For example, when inserting 6, we would follow the tree down the same path as we did in searching 5. Then, because $6 > 5$ we would want to go right, but 5 has no right child yet. This means that a new node with 6 can be inserted exactly at this spot, leading to the situation in Figure 12 (middle).

Removing from a BST is a little more complex, as we might want to remove a node that is right in the middle of the tree "orphaning" its children. For example, let us assume we want to remove 2 from the BST in Figure 12. This would orphan the 1 and the 5 node. It is not possible to simply bring up one of them, as this may violate the BST properties. This can however be fixed by:

- bringing up the left-most of the right descendants of 2, i.e., the smallest value in the subtree rooted by the right child of 2 (i.e., 3).
- Of course this may orphan the right child of this value (3 still has 4 as a right child), but because it is the smallest value of its subtree, it cannot have a left child.
- Therefore, this right child can simply move up to replace the place of this smallest value (i.e., 4 replaces 3).

The removal of 2 is illustrated in Figure 13. Note that for removal, it is useful to keep a `parent` pointer in each node, as will as pointers to the left and right child nodes.

When considering the complexity of *search*, *insertion* and *removal* from BSTs, we see that we always have to take a path down the tree, and possibly do some (constant time, i.e., $O(1)$) shuffling of pointers. Therefore, the dominating factor in the complexity of these algorithms is the number of comparisons we need to do, i.e., the length of the path

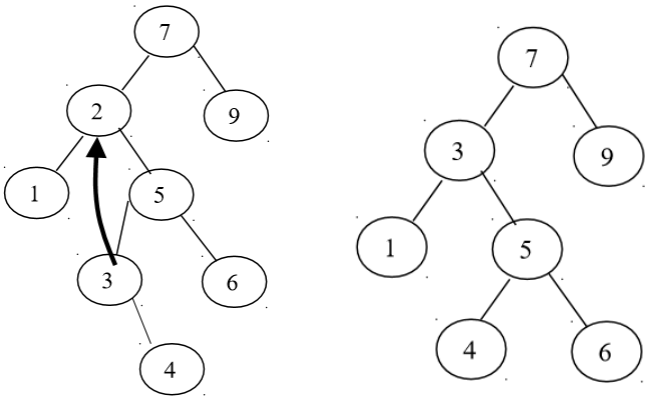

Figure 13: An example of a removal binary search tree: (left) we want to remove 2. (right) after removing 2. Figures by Joop Kaldeway.

we need to take down the tree. This is limited by the *depth* of the tree: i.e., the longest possible path from the root node down to a leaf plus one. For example, in Figure 13 (right) the longest path from the root (7) to a leaf (1, 4, 6 or 9) is of length 4, i.e., $7-3-5-4$ or $7-3-5-6$.

Because the complexity of the operations insert, remove, and search in a BST, $\mathcal{T}$, are all $O\left(\texttt{depth}(\mathcal{T})\right)$, we need to be careful about controlling this depth. For example, from the values in the tree in Figure 13 (right), i.e., $1, 3, 4, 5, 6, 7, 9$, we could also build a BST with depth 3 rather than 4 (can you see how?). We can also build an extremely unbalanced tree by inserting all the nodes in ascending order; leading to 1 at the root, 3 as its right child, 4 as its right child, and so on. The depth of the tree thus depends on (the quite possibly rather random) order in which the values were inserted into the BST. On average, this will often more-or-less balance out the BST in practice. However, of course there are no guarantees. If we do want guarantees, we are going to have to algorithmically ensure balance during insertion and removal.

8.4. Well-balanced Trees

If we want a runtime guarantee in terms of the number of items in a BST for search, insertion and removal, we need to ensure balance in the tree. As the time complexity of these methods for BSTs is $O\left(\texttt{depth}(\mathcal{T})\right)$, and the depth of a well-balanced tree, $\mathcal{T}$, is $\texttt{depth}(\mathcal{T}) = \log n$,[19] we can obtain a time complexity of $O(\log n)$ for all these three key methods. Provided of course that we can do the rebalancing operations in either $O(1)$ or $O(\log n)$ time.

First, let us define what it means to be balanced:

- For all nodes in the tree, the difference in depth of the subtrees rooted by its left and its right child, is at most 1.

[19] Where n is the number of nodes/values in the BST.

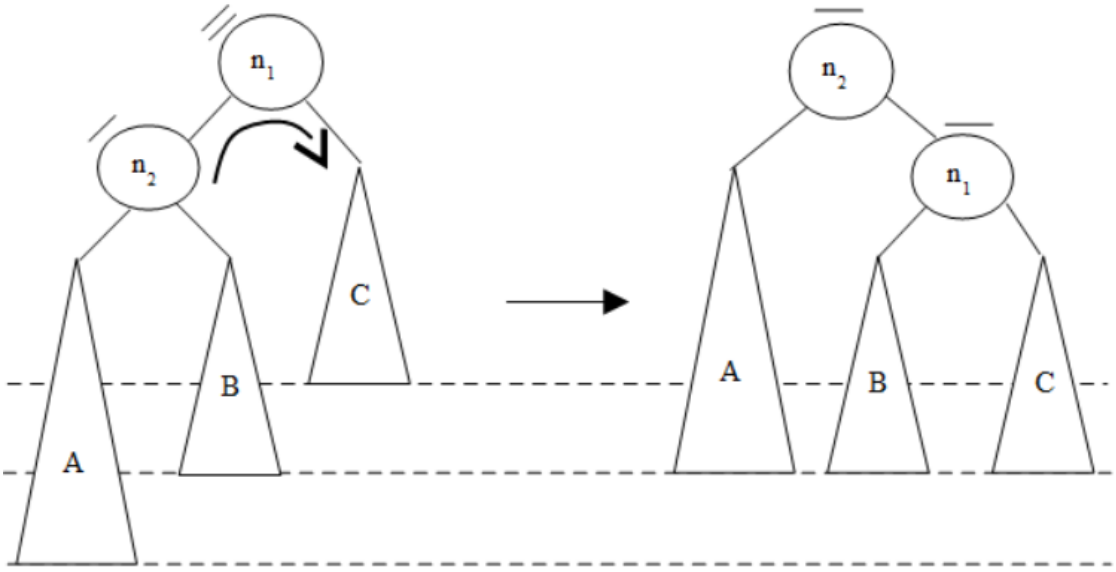

Figure 14: An unbalance in an AVL tree requiring a single rotation. A, B, and C represent subtrees of multiple nodes (not explicitly shown). Figure by Joop Kaldeway.

This means that we have to count the depth of all subtrees rooted by any node in the tree. The depth of a subtree $\mathcal{T}_{\mathcal{N}}$ rooted by a node, $\mathcal{N}$ is:

$$\texttt{depth}(\mathcal{T}_{\mathcal{N}}) = 1 + \max\{\texttt{depth}(\mathcal{T}_{\texttt{leftChild}(\mathcal{N})}), \texttt{depth}(\mathcal{T}_{\texttt{rightChild}(\mathcal{N})})\},$$

where $\texttt{leftChild}(\mathcal{N})$ is the left child node of $\mathcal{N}$, and $\texttt{righttChild}(\mathcal{N})$ is the left child node of $\mathcal{N}$. When $\mathcal{N}$ has no child nodes (i.e., is a leaf), then its depth is 1.

A tree can become inbalanced both by insertion (i.e., its deepest branch becomes 1 deeper) and by removal (i.e., its shallowest branch becomes one shorter). This results in one of two situations, depicted on the left of Figures 14 and 15.

There are two operations necessary for re-balancing a BST when it becomes unbalanced due to an insertion or removal. These were invented by Adelson-Velsky and Landis in 1962 [1]. As such, BSTs that implement automatic rebalancing are now known in the literature as AVL trees.

The first operation, i.e., the *single rotation*, is required when the deepest part of the tree, is on the same side of the child node which causes the imbalence. For example, in Figure 14 (left), node n_1 is unbalanced, as its left subtree (rooted by n_2) is 2 deeper than its right subtree C. Furthermore, the deepest subtree under n_2 is also to the left. In this case the tree can be re-balanced by making n_2 the root, with its left subtree, A, to the left, and n_1 to the right, which now receives B as its left subtree instead of n_2. The results of this *single rotation* is shown in Figure 14 (right).

The situation becomes more difficult when the imbalance is not on the same side as

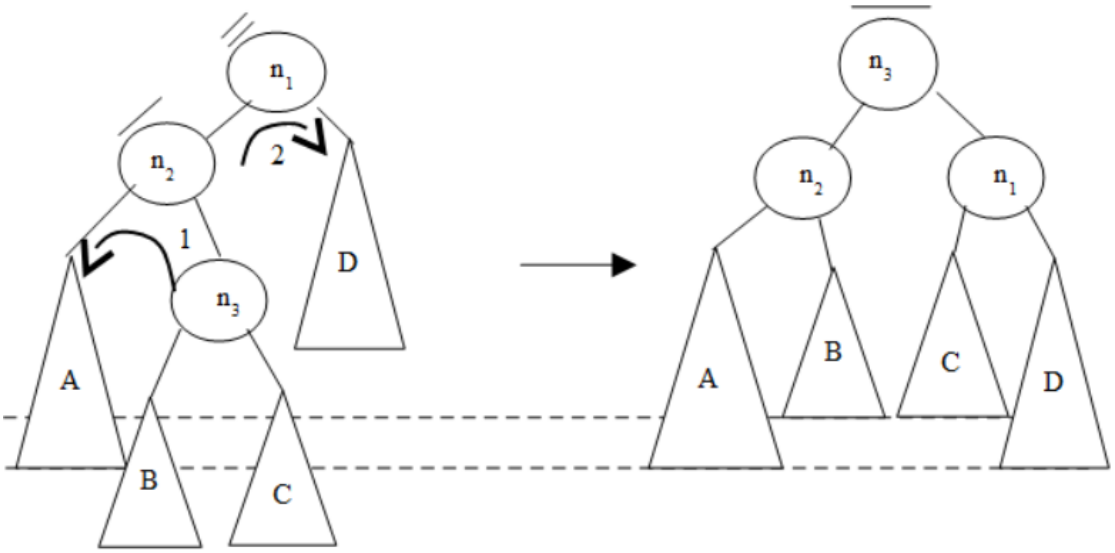

Figure 15: An unbalance in an AVL tree requiring a double rotation. A, B, C, and D represent subtrees of multiple nodes (not explicitly shown). Figure by Joop Kaldeway.

the child node causing the imbalance. For example, in Figure 15, the tree is imbalanced because the left subtree of n_1, rooted by n_2 is deeper than n_1's right subtree (D). However, the deeper subtree under n_2 is now to the right of n_2, i.e., the subtree rooted by n_3. In this case, the tree can be rebalanced by first rotating n_3 to the left (and up), and then again to the right (and up again), resulting in Figure 15 (right). Of course, it is not necessary to implement this as two rotations, one can simply do all the administration of the pointers necessary in one go.

We observe that both the single and the double rotation require a reshuffling of a constant number of pointers. Therefore, the time complexity of these operations is $O(1)$. This may thus be a bit of extra work while inserting or removing, but will not negatively impact the time complexity of either operation. Therefore, in AVL trees the time complexity of search, insertion and removal, are all $O(\log n)$, where n is the number of nodes/values in the tree.

As an exercise, try writing out the pseudocode for insertion, removal, and search in AVL trees. You can show this to your klassendocent to check your answers.[20]

9. Randomised Algorithms

Now let us take a step back from data structures, and move back to algorithmics. As you may have noticed, the previous sections contain a lot of relatively elegant algorithms

[20] Pro tip: we might ask something similar on the exam.

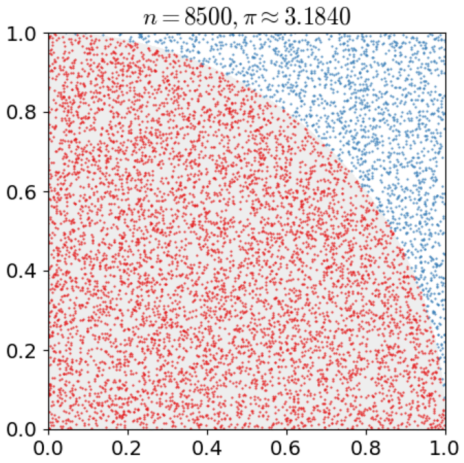

Figure 16: A randomised approximator for ($\frac{1}{4}$ of) π, by drawing random positions between $(0, 0)$ and $(1, 1)$. With probability $\frac{\pi}{4}$ these points (red) will be inside of the quarter circle, and with probability $1 - \frac{\pi}{4}$ outside of it (blue). Figure created by nicoguaro, and taken from `https://commons.wikimedia.org/wiki/File:Pi_30K.gif`.

that have good expected runtimes, but bad worse-case runtimes.

9.1. Randomisation as a stochastic fix

Such worst-cases often depend on a very specific structure in the input. For example, inserting a list of already sorted elements into a BST. Such worst cases tend to not occur out of the blue. For example, humans are a typical source of structure; e.g., they might sort things while inputting the data in the system.

An obvious and often-used strategy to make worst-case input unlikely is to randomise. For example, in the case of adding lists of elements into a BST, we might add these elements in a random order. This would make the worst-case occurring naturally increasingly unlikely (but of course not impossible). This type of randomisation is often applied as a quick fix for worst-case occurrence in production code, and when you encounter it in practice, worst-cases occurring a lot during operation is probably the reason for it.

9.2. Approximation

Another very useful trick using random numbers is approximation for unknown quantities. For example, we can approximate π by drawing tuples of two random numbers, x and y between 0 and 1 (Algorithm 20). These two numbers would correspond to a position on a square with a height and width of 1 (see Figure 16). The probability that $\sqrt{x^2 + y^2} \leq 1$, i.e., that the point is inside the quarter circle shown in the figure (with radius 1), is $\frac{\pi}{4}$. To estimate $\frac{\pi}{4}$, one therefore has to simply count the number of random

positions drawn that are inside this quarter circle N_{in}, and divide this by the total of positions randomly drawn ($N_{in} + N_{out}$).

Algorithm 20 Monte-Carlo Approximation of π

Input: a number of trials, N
Output: an approximate value for π

1: $N_{in} \leftarrow 0$
2: **for** $i \in 0 \dots N-1$ **do**
3: $x \leftarrow$ a random float between 0 and 1
4: $y \leftarrow$ a random float between 0 and 1
5: **if** $\sqrt{x^2+y^2} \leq 1$ **then**
6: $N_{in} \leftarrow N_{in} + 1$
7: **end if**
8: **end for**
9: **return** $4 \cdot \frac{N_{in}}{N}$

The above procedure, which uses random number generation to estimate something that is actually deterministic, is an example of Monte-Carlo method.[21] Monte-Carlo methods are used widely to estimate all kinds of quantities of static objects as well as processes and decision problems. For example, they have even been applied in theoretical physics [13]. In this reader, we will be looking at its application – together with trees (Section 11) – to optimise policies/strategies for automated game-playing in Section 12.

10. Intermezzo: Automated Game Play → Optimisation

We have seen multiple classic algorithms and data structures that have been applied in almost every piece of software (and hardware for that matter) since the beginning of computer science, i.e., inserting, sorting, removing and searching, in different kinds of container data structures. These are essential bits of knowledge that will serve you well in designing many an application for a variety of platforms, as long as you remember to always think which operations are key to the performance of the system, and reduce the time complexity of those operations accordingly.

For the classic problem types above, we do not expect much more improvements to follow in the future above the old faithful algorithms already described. What we want to do in the rest of this course is give you an insight into modern algorithmics, on problems where research is still ongoing. A – par chance – highly suitable domain to study these algorithms is of course *gaming*. This is because games are things we as humans have invented for ourselves to challenge our own intellect (among other things of course). As such these games provide highly challenging domains for automated play.

Before we go into *automated game play* however, it should be noted that while automated game play may seem highly specific, it is itself an instance of *sequential decision*

[21] Yes, indeed, this is named after the Monte Carlo casino in Monaco.

making. Sequential decision making is of course a much more general problem, that applies to planning under uncertainty for train parking and maintenance [14] or mapping a program onto a (dedicated) chip [15]. Furthermore, sequential decision making itself is an instance of *optimisation*. Optimisation is a highly general problem which says: we have to input a number of variable values, i.e., an array (or vector) of inputs, $\mathbf{x}$, and given the optimisation problem, there will be some evaluation of this to a reward, $f(\mathbf{x})$. An optimisation task is then simply $\max_{\mathbf{x}} f(\mathbf{x})$.[22] That is, find such inputs that the reward is maximised. This can of course be very complex: in automated game play, $\mathbf{x}$ is an entire specification of how we want our program to play the game, and $f(\mathbf{x})$ can be our expected probability of winning the game.

The algorithms we will present in the next section are general sequential decision making algorithms, but can (with a bit of adaptation) also be used in wider optimisation tasks.

11. Tree Search Algorithms

Please note that this section is written specific to games. Of course, a general sequential decision making description would not be much different.[23] *A general optimisation description would be a bit different.*

What does it mean to play a game? Basically, a *player* gets a state of the game, i.e., a *game state* as input, and then makes a *move*. In other words, the player needs a *policy*, i.e., a specification of what move to make in any valid game state that the player might encounter. Automated game play therefore means optimising such a policy as to maximise the probability of winning.[24] There may also be another player, who might be fully adversarial to your player. For example, if you play naughts in tic-tac-toe (boter-kaas-en-eieren), the crosses player will try to minimise your probability of winning (and thus win himself instead). In such games, a move by a single player is also called a *ply*.

A natural way to model a policy in a game is to build a game tree (Figure 17). The root of this tree represents the begin (or current) state. From this game state, other game states can be reached by doing moves. If the game is deterministic (i.e., the next state is always the same after playing a given move in a given game state and does not depend on any randomness), the number of children of a node in the game tree is thus equal to the number of valid moves the player can do.

The best move is the move that maximises the score (i.e., do we win?). Of course, this can only be determined by going down the tree: I maximise my score by taking the best move for the current position, then my opponent will take the worst possible move for me, then I maximise again, and so on until the end of the game. While building the game tree, it is most efficient to start at the leafs, and then *backup* the value of these

[22] Or, if the rewards depend on a stochastic process (i.e., have a degree of randomness), it should be written as the expected reward given the input $\mathbf{x}$, i.e., $\max_{\mathbf{x}} \mathbb{E}(f(\mathbf{x}))$.

[23] The term *game state* become *(environment) state*, *players* would become *agents* and *plies/moves* would become *actions* or *decisions*.

[24] Or to maximise the score, or some other metric of successful game play. This is a design choice.

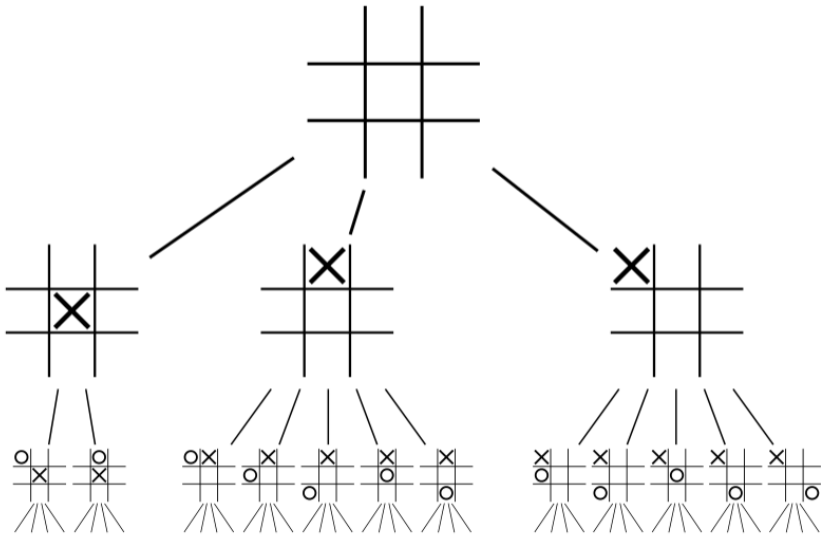

Figure 17: Game tree for tic-tac-toe. Please note that only unique positions after symmetry-based reduction have been inserted into this tree. Figure taken from `https://commons.wikimedia.org/wiki/File:Tic-tac-toe-game-tree.svg`, own work by en:User:Gdr.

leafs to their parents. That is, the parent node will take the maximising child as its own value (or minimising if it is the opponent's turn).

On possible issue in building a full game-tree is the size of this tree. For example, if we take the game of tic-tac-toe, there are 9 valid moves on an empty field, leading to 9 different game-states. Then, the subsequent player can do 8 different moves, and so on. This would mean that the full game-tree would consist of $9! = 362880$ nodes. This is of course still very doable for a computer. Note however that in more complex games, e.g., *go*, which is also turn-based but has 19×19 fields, this will most likely no longer be doable. In this case, limited gain be achieved by exploiting the fact that in many games (and other decision problems), some states are actually the same after mirroring or rotation, and this should not matter for their value. For example, in tic-tac-toe, when thinking about it in a structured manner, the three options for the first move are actually: the middle, one of the sides, or a corner, and it does not actually matter which side or which corner you choose. This leads to a reduced game tree of which the first two layers are displayed in Figure 17. For most games however, we need to accept that we will never be able to systematically search the entire game tree.

12. Monte-Carlo Tree Search

Many games have too many (reachable) game states for algorithms to generate a entire game tree for. Therefore, to find out what the best move to play currently is, we need to

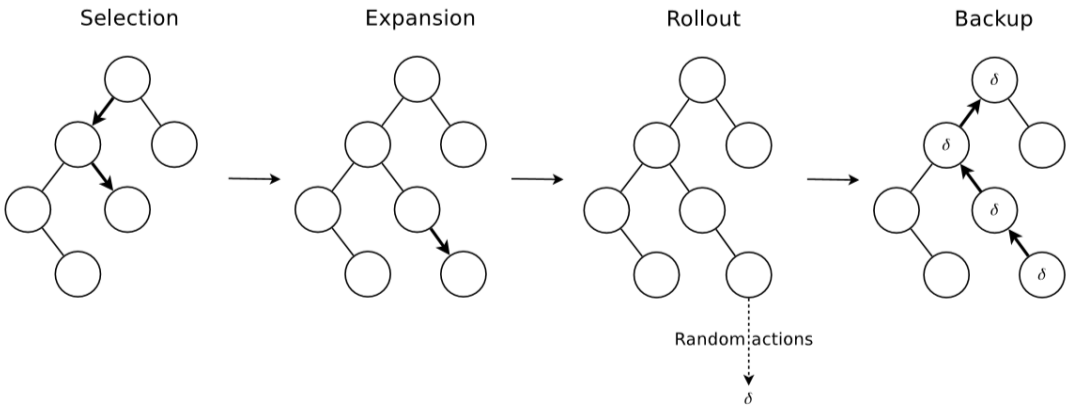

Figure 18: The basic steps in one iteration of Monte-Carlo Tree Search. Image taken from De Waard, Roijers and Bakkes (2016) [4].

rely on approximate (rather than exact) algorithms. What we can do instead of exploring this entire game tree, is exploring different parts of the game tree in a well-reasoned, but fundamentally heuristic manner. This has led to one of the most well-known algorithms in automatic game-playing: *Monte-Carlo Tree Search (MCTS)*. In this section we start with a short overview of the algorithm, after that we dive in more detail in the algorithm. If you find it easier to first look at the pseudo-code then go to Algorithm 21.

Instead of building the entire game tree, MCTS builds a partial tree (left). Of course this means that MCTS cannot know exactly what the value (i.e., whether it can win or will lose) for each game state is, as it is no longer able to maximise over our moves and minimise over the opponent's moves until the game ends. Therefore, we need some kind of stand-in estimate of the value of a game state. For this, we use randomisation (Section 9).

In this partial tree, the leaves correspond to game states that may not be terminal (win or lose). To estimate the value of such game states, MCTS performs a lot of random games (Monte-Carlo roll-outs) using a random (or better heuristic) policy for both players from that game state onward. Playing such a random game until termination (i.e., one of the players wins, or there is a draw), is called a Monte-Carlo *roll-out*.

Now that we have a way to evaluate leaf nodes (game states) that do not yet correspond to finished games, MCTS can build its partial tree iteratively. That is, it starts at the current (or begin) game state, and add child nodes (representing subsequent game states after a single move), as in standard full tree search. However, when we add a child node, we now estimate its value by a lot of random plays. Adding a new child node is called *expansion* of the game tree.

After *expanding* and evaluating the leaf node through *roll-outs*, the new leaf node gets a value. This value needs to be processed back up the tree. This is called *backup*. After the values have been back-upped, MCTS can select a new place to extend the game tree with a new node. For such *selection*, we need to balance exploration, i.e., looking at

previously un(der)explored possible moves, and exploitation, i.e., further investigating moves that we already highly suspect that they are good. Together these steps form an iteration of the main loop of MCTS: 1) select a place to expand the tree, 2) create the new node, 3) perform Monte-Carlo rollouts to determine the value of the node, and 4) back up this value to the ancestors of the new node (i.e., its parent, its parent's parent, and so on).

Algorithm 21 Monte-Carlo Tree Search

Input: a current game state s_0
Output: an action/move a

1: $n_{root} \leftarrow$ create the root node with game state s_0
2: **while** Not out of time **do**
3: $n_{leaf} \leftarrow$ `FindSpotToExpand`(n_{root}) ▷ Algorithm 22
4: $val \leftarrow$ `rollout`(n_{leaf}) ▷ Algorithm 23
5: `BackupValue`(n_{leaf}, val) ▷ Algorithm 24
6: **end while**
7: **return** the move/action corresponding to the best child, ch, of n_{root} (that is, with the maximal value for $\frac{ch.Q}{ch.N}$)

12.1. (New) Nodes

Now that we have seen the basic overview of the algorithm, let us work out the details. First, let us discuss which attributes a node in Monte Carlo Tree Search has. Subsequently, we discuss how these attributed are updated.

A node in MCTS corresponds to a given game state (e.g., a position on a board of go or gomoku). For conceptual ease, we assume that the game we are playing is deterministic, i.e., given a game state a specific move always leads to the same subsequent game state. A node in the search tree in MCTS requires the following attributes:

- A parent (pointer, for ease of use, corresponding to the previous game state)
- A container with children, corresponding to possible moves/subsequent game states.
- a number N of visits to the node – this is used for exploration purposes
- a value Q, the total number of accrued points, i.e., the number of wins plus 0.5 times the number of draws.

Please note that this list is not exhaustive. For example, it can make a lot of sense to store the the game state (for example, in gomoku this would be a board and whether it is black's or white's move). You can add anything you which as attributes of the nodes.

The value is (typically) the number of wins recorded for roll-outs after this game state (corresponding to this node). This is different from vanilla Tree Search with a complete game tree: when we can complete the game tree, we can determine exactly whether we can win in a given game state. In contrast, MCTS depends on random roll-outs. This is

an imprecise way of estimating the value. Therefore, it makes sense to take the average probability of winning for every roll-out that includes the current game state. By doing so the estimated probability of winning depends on many roll-outs, and is therefore a better estimate.

Algorithm 22 `FindSpotToExpand`

Input: a node, n, in the tree
Output: a new node in the tree (unless a terminal node visited)

1: **if** n is terminal (game finished) **then**
2: **return** n
3: **end if**
4: **if** n is not fully expanded **then**
5: $n' \leftarrow$ a new child node for a not-yet-explored move (with n as its parent)
6: add n' to n's children
7: **return** n'
8: **end if**
9: $n' \leftarrow$ child with the highest `uct` value ▷ Equation 1
10: **return** `FindSpotToExpand`(n')

First let us look at how to expand the game tree (the first step in the loop of Algorithm 21) based on Q and N. Algorithm 22 recursively finds a path through the (current) game tree, until it hits a node that is not yet *fully expanded*. A node is fully expanded when all valid moves in the game state corresponding to this node, have been added as a child node. In other words, a node is fully expanded if its number of children is equal to the number of valid moves in that game state. Once a node is found that is not fully expanded, a random not-yet-explored move is selected to add as a child node.

If all possible children have been added to a node n (i.e., all valid moves have been visited), we need to select one of n's children to further inspect. There, we need to balances exploitation (we want to visit nodes that we already expect to be good), and exploration (we want to visit nodes that have not been visited often yet. Kocsis and Szepesvári's [8] original proposal uses a theoretical so-called upper-confidence bound to do so:

$$\texttt{uct}(n) = \frac{\texttt{n}.Q}{\texttt{n}.N} + c\sqrt{\frac{2\ln \texttt{n.parent}.N}{\texttt{n}.N}}. \tag{1}$$

The first part, $\frac{\texttt{n}.Q}{\texttt{n}.N}$, is just the number of wins (or losses in case of the opponent, see Algorithm 24) divided by the total number of visits. In other words, this is the exploitation term. $c\sqrt{\frac{2\log \texttt{n.parent}.N}{\texttt{n}.N}}$ looks at how often if n 's parent is visited $\texttt{n.parent}.N$ versus how often this node is visited $\texttt{n}.N$. Because exploring is very important when there are few visits, but quickly becomes less important as more data is gathered, the natural logarithm (ln, i.e., the e-log) is taken over the number of visits to the parent. This is because logarithms grow very slowly (see Figure 2), so when we divide it by the number of visits, it quickly becomes small. c is a constant that can be used to fine-tune the exploration.

The original paper by Kocsis and Szepesvári's [8] uses $c = 1/\sqrt{2}$, in order to prove that the regret (roughly, how often bad decisions are taken), is limited. However, you can play with the value of c in order to change the strength of exploration. Furthermore, the UCT-value can be replaced by a different exploration mechanism all-together. For Go for example, Coulom's *crazyStone* formula [3] has been shown to be effective. In the practical assignments, you can try different options for the game of *gomoku* that you can find in the literature, or even invent your own.

12.1.1. Monte-Carlo rollouts and Backing-Up the values

Once MCTS has found a node to expand, it created a new child node. This new child node corresponds to a game state. If this game state is non-terminal (i.e., the agent has not won or lost yet), MCTS performs a roll-out (a random game) with a random policy. This is shown in Algorithm 23.

Algorithm 23 `Rollout`

Input: a node in the tree n, corresponding to a game state s
Output: a reward (typically win (1), loss (0), or draw (0.5)

1: **while** s is not terminal (game ongoing) **do**
2: $a \leftarrow$ random action/move in state s
3: $s \leftarrow$ the new game state after executing a
4: **end while**
5: **return** result of the game (e.g., 1 if win, 0 if loss, and 0.5 if draw)

After obtaining the result of a roll-out, MCTS processes the result of this roll-out back up the tree (Algorithm 24). Note that here we need to perform a small trick; our agent is trying to win, but so is the opponent. Because if our agent wins, the opponent loses, MCTS can multiply the value by -1 when the opponent is to play[25]. By doing so, maximising the value in a game state where it is the opponent's move results in the best move for the opponent (as we would like).

Algorithm 24 `BackupValue (2-player game)`

Input: a result for a game `val`, and a node n

1: **while** n is not `null` **do**
2: $n.N{+}{+}$ ▷ Increase the number of visits
3: **if** n corresponds to a game state where the opponent is to move **then**
4: $n.Q \leftarrow n.Q -$ `val` ▷ because the opponent minimises the agent's value
5: **else**
6: $n.Q \leftarrow n.Q +$ `val`
7: **end if**
8: $n \leftarrow n.parent$
9: **end while**

[25] Or as i shown in Algorithm 24, we can subtract the value instead of adding it.

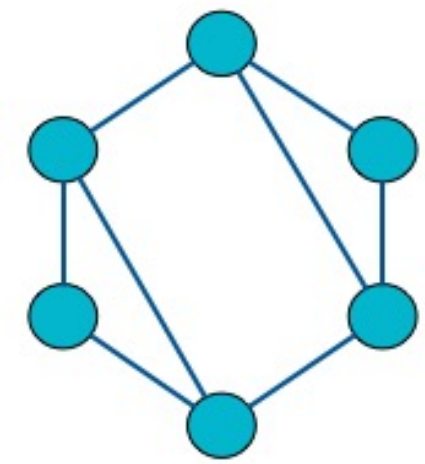

Figure 19: An example of a graph with 6 nodes (circles), and 8 edges (lines between the nodes). Figure taken from `https://commons.wikimedia.org/wiki/File:2-edge-connected.jpg`, own work by Tmigler.

In this section, we have assumed a completely random policy to perform the roll-outs. It has been shown in the literature however, that being quite a bit less random can be more effective; i.e. using a more intelligent strategy for the rollouts can help. However, if this strategy is not (always) good, this might have averse effects. Furthermore, if the policy takes up too much computation time, it will eat into the computational budget of generating more nodes to explore. So, while this can help a lot – and we encourage you to try in the practical assignments – the more informed strategy needs to be constructed carefully.

13. Graphs

Until now, we have been looking at fully (array, list), and partially (trees) ordered data structures. Trees for example are partially ordered, as there is a clear parent-child node ordering, but there is no order between the child nodes. *Graphs* are different. A graph is a data structure that models relations between elements only, without order. A good example of this is a road network. Two locations, A and B, are directly connected if there is a(n uninterrupted) road from A to B, but there is no clear order between different locations

In graphs, we have *nodes* (also called vertices) that are connected by *edges*. Graphically, graphs are typically represented as circles for the nodes, and lines between them for the edges (as in Figure 19). Graphs are a prevalent data structure for topological data (such as the road network), and are therefore commonly used in e.g., SatNav systems (navigatiesystemen). However, there are many other types of data (and associated problems) for which graphs are a useful data structure:

- In *planning problems* (e.g., in a factory), nodes can represent different tasks. If two tasks require the same resource (e.g., the same machine), they cannot be planned in the same time slot. (Other examples of such scheduling of tasks occur on executing a program (with subroutines) on dedicated chips [15].)
- When you are playing robot football[26] [9], the different robots (nodes) can only directly influence other robots that are close (in the next timestep). The rewards such local pairs/groups can be expressed as edges.
- Social networks can be expressed as graphs. Each person is a node in the graph, and there is an edge between two people in the graph that have a relation. This can be being friends, or playing a game on the same team [18], or just meeting face-to-face and thus having the opportunity to spread influenza viruses to each other [2].
- In probability networks, the nodes on a graph are variables, and the edges probabilities on their joint values. This is for example used in image analysis and editing, where two adjacent pixels are likely to have similar colour values.

Graphs are typically represented as a tuple: $G = (V, E)$ where V is the set of nodes (vertices) and E is the set of edges. Edges are typically represented as tuples of nodes. So if $(1, 2) \in E$, it means that there is an edge between node 1 and node 2. Both nodes and edges can contain additional data. Of course, this can be implemented in many different ways. You will see just one of these possible implementations in the practical assignments.

There are a couple of choices in the specification of graphs that have an impact on their implementation. Specifically, many graphs are *undirected*, meaning that if there is an edge from node 1 to node 2, the reverse (there is an edge from node 2 to node 1) is also true. in *directed* graphs, $(1, 2)$ and $(2, 1)$ are different edges. This is for example important for road networks, where some roads may be one-way only. For example, in Figure 20 it is possible to "travel" from A to B, but is not possible to travel from B to A.

Furthermore, in most graphs, edges are only possible between exactly two nodes. This can be generalised to hyperedges, that can make a connect between any number of nodes, e.g., an edge (in this case also called a *factor*) $(1, 2, 5)$ laying a connection between three nodes (1, 2, and 5). The resulting data structure can be referred to as a *hypergraph* or *factor graph*. In such graphs, nodes are usually depicted as circles, but hyperedges (factors) are typically represented as squares, connected with lines to the nodes. Factor graphs are for example useful in the case of planning problems with resources (as above), where more than two tasks need the same resource, or to model rewards in multi-agent decision problems (such as robot football).

[26] We respectfully decline to call it robot "soccer".

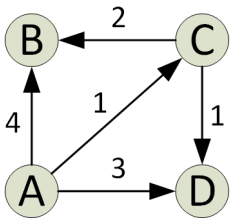

Figure 20: An example directed graph with 4 nodes and 5 (directed) edges. In addition to the connection, the edges also contain a number, e.g., representing the travel time (in minutes) when taking this (unidirectional) node). Figure is own work by Sanoj68, and was taken from `https://commons.wikimedia.org/wiki/File:Apsp_dijkstra_graph.png`.

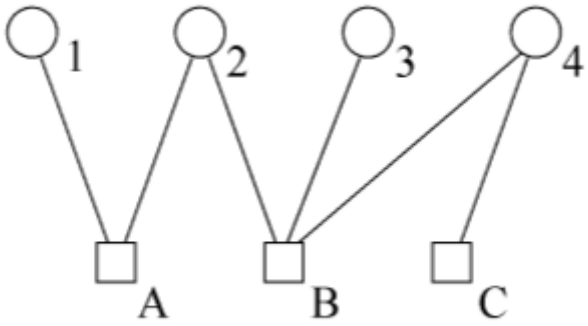

Figure 21: An example factor graph with four numbered nodes, and three labelled factors. For example, the edge $B = (2, 3, 4)$ connects three nodes. Figure taken from `https://commons.wikimedia.org/wiki/File:Factor_Graph_representation.png`, own work by 14ndy15.

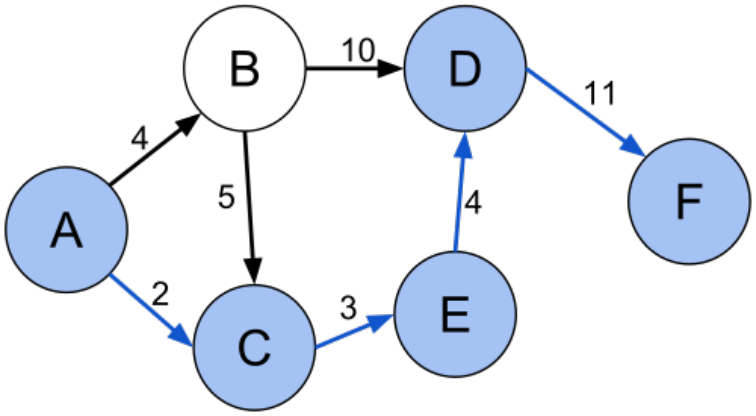

Figure 22: An example of the shortest path problem expressed as a graph. The shortest path from A to F is coloured blue. Image taken from `https://commons.wikimedia.org/wiki/File:Shortest_path_with_direct_weights.svg`, own work by Artyom Kalinin.

14. Shortest Path

Now let us discuss a particular problem often modelled as a graph: the shortest path finding problem. We get an undirected graph as input. The nodes in the graph represent locations. The edges represent stretches of road between these locations. The nodes are labelled. The edges, e.g., (A,B) contain a travel time (in minutes) that it takes to travel the stretch of road (e.g., from A to B). Given two locations/nodes in the graph, `start` and `finish`, how can we get from `start` to `finish` in the minimal possible travel time? An example of this problem, with its solution is provided in Figure 22.

14.1. Dijkstra's algorithm

One of the most famous algorithms for solving the shortest path problem is Dijkstra's algorithm. This algorithm is named after Edsger Wybe Dijkstra, who first published the idea for this algorithm in 1959 [5]. His original description of the algorithm was done in less than two pages, entirely in words; it is a highly elegant algorithm.

Observation (1) The algorithm starts from the following observation: if we know that a node R is on the shortest path from node P to node Q, this means that the shortest path from P to Q contains the entire shortest path from P to R. In other words, to find the shortest path from P to Q, we must know what the shortest path from P to R is.

Observation (2) Further more, if we know that the shortest path from A to C is at most length l, e.g., $l \leq 2$ in Figure 22, and we know that the shortest path from A to B is $k > l$, e.g., 4 in the figure, then the shortest path from A to C cannot contain B, as a path from A to C via B would be at least k in length. This observation depends on an

upper bound, i.e., the shortest path from A to C is at most l. These upper bounds are relatively easy to establish; the length of any path from A to C is an upper bound on the length of the *shortest* path from A to C.

These two observations to build an iterative algorithmic scheme. I.e., we are going to create a set of nodes for which we have established the shortest path from the `start` already, let us call this set $\mathcal{S}$ for "solved". A set of nodes nodes that are not yet solved, but have an edge from at least one node in $\mathcal{S}$ directly to these nodes. Let us call this set $\mathcal{N}$, for "neighbours". Finally, there is the rest of the nodes $\mathcal{R} = (V \setminus \mathcal{S}) \setminus \mathcal{N}$. In every iteration the algorithm will:

- move exactly one node from $\mathcal{N}$ to $\mathcal{S}$, i.e., finding the shortest path from `start` to one new node in the graph,
- move any number of nodes from $\mathcal{R}$ to $\mathcal{N}$, i.e., selecting new candidates to compute a new shortest path for, and
- compute a new upper bound for the distance for any number of nodes in $\mathcal{N}$.

Dijkstra's algorithm is given in Algorithm 25. Each node initially gets an upper bound on their distance to the starting point of infinity (except for the starting node itself, which gets 0). At each iteration, the algorithm visits a new node in $\mathcal{N}$. Specifically, the node from $\mathcal{N}$ with the lowest upper bound on their distance is taken from $\mathcal{N}$. For this node, we will be sure that there is no shorter path to it from the start than its current upper bound. In other words, its current upper bound is the shortest path. This is because when the algorithm visits a node, it uses this node to adjust the upper bounds of its neighbouring nodes (all nodes directly connected to it by an edge). I.e., is there a path through the current node to one of its neighbours that is shorter than their current upper bound. Because we always select the node with the smallest upper bound, its upper bound results from an already solved node plus its distance to that solved node, and the fact alternative paths would have to come through a different unvisited neighbour whose path can only be longer (as we have checked all solved nodes in previous iterations), the current upper bound must be the shortest path.

Algorithm 25 `ShortestPath`

Input: A graph $G = (V, E)$, a `start` node, and a `finish` node.
Output: The shortest path from `start` to `finish`

Set `n.dist` to ∞ (infinite) for all nodes in $n \in V$
Set `n.prev` to `null` for all nodes $n \in V$
`start.dist` $\leftarrow 0$
$\mathcal{N} \leftarrow$ a new set with only `start` in it
$\mathcal{S} \leftarrow$ an empty set
while $\mathcal{N}$ is not empty **do**
    $n \leftarrow \arg\min_{m \in \mathcal{N}} m.dist$ ▷ select node with minimal distance in $\mathcal{N}$
    move n from $\mathcal{N}$ to $\mathcal{S}$ ▷ remove from the former, add to the latter
    if $n =$ `finish` **then**
        `break`
    end if
    for neighbour m of n for which $m \notin \mathcal{S}$ **do**
        if $m \notin \mathcal{N}$ **then**
            add m to $\mathcal{N}$
        end if
        altDistance $\leftarrow$ `n.dist` + (`n`, `m`).`length` ▷ NB: (`n`, `m`) is an edge
        if `m.dist` $>$ altDistance **then**
            `m.dist` $\leftarrow$ altDistance
            `m.prev` $\leftarrow n$
        end if
    end for
end while
return `finish.dist` and the shortest path ▷ Computed recursively by visiting the `prev` element from `finish` to `start` and adding them to a list in reverse order

Dijkstra's algorithm is a highly effective algorithm. When we analyse its runtime, we find that it (can) loops through all nodes with the while loop (as long as there is at least one path from one node to every other node in the graph). In each of this iterations, the algorithm must find the node in $\mathcal{N}$ with the minimal current upper bound on the distance. When $\mathcal{N}$ is implemented via a simple linked list (note that this might be useful, because we remove one node every iteration, and possibly add multiple ones) computing this minimum is what becomes the dominating factor, as it would require us to loop through the list. Because $\mathcal{N}$ can at any time contain at worst all nodes in the graph (or at least a high proportion of them), the time complexity of such an implementation of Dijkstra's algorithm would be $O(|V|^2)$, where $|V|$ is the size of the set V, i.e., the number of nodes in the graph.

So can this be made faster by choosing a more efficient way of storing $\mathcal{N}$. Specifically because we want to find the minimal distance every iteration, we ought to be able to use a priority queue for this. However, a snag here is that we not only add new nodes to the priority queue, we also have current nodes in this priority queue for which the

priority may change (and therefore so should their place in the queue). This leads to unfavourable worst-case time complexity results. This problem was finally solved in 1984 (later more extensively described in a 1987 journal article) by Fredman and Tarjan [6], with the introduction of the Fibonacci Heap data structure.[27] This reduced the time complexity to $O(|E| + |V| \log |V|)$, where $|E|$ are the number of edges and $|V|$ are the number of nodes. [28]

14.2. Dynamic programming for finding shortest paths

An alternative to Dijkstra's algorithm, and another algorithm that is very well known in the literature is *value iteration (VI)* which falls in the class of so-called *dynamic programming (DP)* algorithms. As the name suggests, the algorithm iteratively improves the value estimate, i.e,. the length of the shortest path to the `finish`, from every other state, i.e., node/location in the graph. We first explain this algorithm in terms of the shortest path problem, but note that this algorithm can be applied to a much broader class of problems. We will discuss the broader applications of the VI algorithm at the end of this section, and DP algorithms in general in Section 15.

VI for the shortest path problem starts from the following observation:

Observation (3) If I am in node n on the graphs, the length of the shortest path from node n to the `finish` is the minimum of the shortest path of the neighbouring nodesplus the length of the edge that takes me to that neighbouring node. In other words,

$$shortest(n) = \min_{m \in \texttt{neighbors}(n)} shortest(m) + (n, m).length, \tag{2}$$

where $(n, m).length$ is the length of the edge between n and m, and $\texttt{neighbors}(n)$ is the set of nodes for which there is a edge in E that directly connects to n. This equation may look like the start of a neat little recursive algorithm, but it is not. This is because there can be *loops* in the graph. Particularly, if for a node n we call the function `shortest` on all its neighbours, then those neighbours will try to call $\texttt{shortest}(n)$ in order to compute it. So that does not work. No trick (not even memoisation) can save us from that.

So we need a different strategy. Specifically, we can use Equation 2 not as a rule for recursion, but as an update rule. That is, if we have an array of current upper bounds for the shortest path from each location to the `finish`, then we can get a set of tighter (i.e,. better) upper bounds by applying the update rule.[29] When we naively apply the update rule, this leads us to Algorithm 26. Please note though that there is an inefficiency in it, see if you can spot it.

[27] Hell yeah, Fibonacci numbers! Remember Algorithms 6 and 7?

[28] Don't worry, we won't ask this last fact on the test, but we encourage you to check out the papers (or the internet) regarding Fibonacci heaps.

[29] In fact, it even works if the initial values are not upper bounds, but indeed any values. This is because we will override the values at every iteration, with more realistic values. As an exercise please verify this for yourself.

Algorithm 26 `Value Iteration (bad) for Shortest Path`

Input: A graph $G = (V, E)$, a `start` node, and a `finish` node.
Output: An array with the length of the shortest path from any node in the graph to `finish`

1: $dist2 \leftarrow$ an array of length $|V|$ filled with ∞'s
2: Give every node an `index` in the array, take `start.index` $= 0$ and `finish.index` $= |V| - 1$.
3: $dist2$[`finish.index`] $\leftarrow 0$ ▷ by definition
4: $dist \leftarrow$ an array of length $|V|$ filled with NaN's ▷ Not a number
5: **while** `dist` does not equal `dist2` **do** ▷ all values
6: $dist \leftarrow dist2$
7: **for** $n \in V$ **do**
8: $dist2$[`n.index`] $\leftarrow \min_{m \in \texttt{neighbors}(n)} dist$[`m.index`] $+ (n, m).length$
9: **end for**
10: **end while**
11: **return** $dist$ ▷ Note that the shortest path can be extracted from $dist$, using $\arg\min_{m \in \texttt{neighbors}(n)} dist$[`m.index`] $+ (n, m).length$ to select which neighbour to go to.

This algorithm loops over all nodes in the graph in every iteration, and for all nodes over all its neighbours. This is quite expensive (especially as in the worst case, all nodes can neighbour all nodes). So the inner loop already has a complexity of $O(|V|^2)$, without even considering the outer (while-) loop. We can in fact do much better, by noting that we can check every edge instead. This leads to Algorithm 27 – known as the Bellman-Ford algorithm – which has just $O(|E|)$ for its inner loop.

Algorithm 27 `Bellman-Ford (better) for Shortest Path`

Input: A graph $G = (V, E)$, a `start` node, and a `finish` node.
Output: An array with the length of the shortest path from any node in the graph to `finish`

1: $dist2 \leftarrow$ an array of length $|V|$ filled with ∞'s
2: Give every node an `index` in the array, take `start.index` $= 0$ and `finish.index` $= |V| - 1$.
3: $dist2$[`finish.index`] $\leftarrow 0$ ▷ by definition
4: $dist \leftarrow$ an array of length $|V|$ filled with NaN's ▷ Not a number
5: **while** `dist` does not equal `dist2` **do** ▷ all values
6: $dist \leftarrow dist2$
7: $dist2 \leftarrow$ an array filled with ∞'s, except $dist2[|V| - 1] = 0$
8: **for** $(n, m) \in E$ **do**
9: $dist2$[`n.index`] $\leftarrow \min\ \{dist2$[`n.index`]$, dist$[`m.index`]$ + (n, m).length\}$
10: $dist2$[`m.index`] $\leftarrow \min\ \{dist2$[`m.index`]$, dist$[`n.index`]$ + (n, m).length\}$
11: **end for**
12: **end while**
13: **return** $dist$ ▷ Note that the shortest path can be extracted from $dist$.

So, how often does the while-loop need to run? We can find a reasoned answer for this. We note that at the first iteration, the shortest path for at least one node to the finish (i.e,. one of its neighbours) will be found. Once the shortest path has been found, its shortest path length estimate in $dist$, will not change in subsequent iterations anymore. Then on the next iteration, the algorithm will find the shortest path for at least one more node, and so on. This can of course only continue for $|V|$ times, as there are only $|V|$ nodes in the graph. Hence, we know that the outer-loop can only run $|V|$ iterations, plus one to confirm that nothing has changed. Note though that this is of course the worst case. The algorithm may well terminate much faster.

Another way to see that the outer loop needs only $|V|$ iterations is by making the following observation: in each iteration, i, the shortest paths to the `finish` that consist of at most i edges (so $i + 1$ nodes), become known. Furthermore, we note that longest possible shortest path connecting `start` to `finish` can use $|V| - 1$ edges; if it would be more, there would be at least one node which would be on the path more than once, which would make no sense. Therefore, the number of iterations is at most $O(|V|)$.

Noting that the outer loop runs for $O(|V|)$ times, and the inner loop takes $O(|E|)$, this brings the runtime for the algorithm to $O(|V||E|)$. In most cases, $|E| > |V|$ so this algorithm seems worse than Dijkstra's shortest path algorithm. Often though, it is not much worse. For example, on road networks, $|E|$ is $O(|V|)$ as well. This is because road networks are so-called *planar* graphs, i.e., graphs that can be drawn on paper (such as a map), without edges crossing each other. For planar graphs, the maximum number of edges is at most $3|V| - 6$.

14.2.1. VI for planning in decision problems

So, if VI is not better than Dijkstra's algorithm for the shortest path (not even after the improvements of the Bellman-Ford algorithm), why did we discuss it? That's because it is in fact a really famous algorithm, that can easily be extended to different problem classes. Specifically, in the shortest path problem, the location (or node) where a person (or agent) currently is, is the state. Furthermore, going to one of the neigbouring nodes via an edge, are the actions it can take in that state. We have seen a similar formulation before in Section 10. In general, we can do the same for any (deterministic) decision problem for which we can observe:

$$value(s) = \max_a reward(s, a) + value(s_{next,s,a}),$$

i.e., that the value of a state, is maximum of the the immediate reward we get for doing action a in state s plus the value of the next state we end up in by doing action a in state s, $s_{next,s,a}$. (Note that we have moved away from costs (such as the travel time in the shortest path problem), to positive rewards. This makes no difference, as costs are just negative rewards.) This means that we can use VI for any type of game in which the next state will deterministically depend on the current state and the action.[30]

However, VI can also be extended to the situation in which an action does not always lead to the same next state. This is true for many games (such as video games), but also for tasks like robot navigation, where wheels can slip, or objects can suddenly move into the way, etc. In this case, the probability of the next state given a current state and an action should be taken into account, but it is still basically the same idea. A further tweak also allows for games and other decision problems that may never end. In other words, VI is a planning algorithm that can be used for a large variety of decision problems. If you want to know more about this see [16] (this is a book that's available online freely at `http://incompleteideas.net/book/the-book.html`) or [10] which also discusses applications in the business domain (e.g., how such an algorithm can be used in the context of deciding the pricing of airline tickets).

15. Dynamic Programming (in general)

Value iteration falls into the larger class of *dynamic programming* algorithms. *Dynamic programming (DP)* algorithms use a divide-and-conquer strategy, similar to what we have seen for recursive algorithms with memoisation, i.e., they both start with the observation of a recurrent relation between larger and sub-problems. For example, for Fibonacci we have the following recurrence relation:

$$fib(n) = fib(n-2) + fib(n-1).$$

From this observation, (memoised) recursive algorithms start at the most difficult problem and work their way back. DP algorithms do the opposite; they work bottom-up;

[30] Also note that we do assume that the game will always end at some point, whatever the player(s) do(es).

they start at the simplest problems they can solve (i.e., the base cases), and then greedily solve larger instances of the problem.

Step 1 To do so, they identify the smallest possible state-descriptor:

Definition 3 *A* state *of a computation in a DP problem, is the set of parameters that can uniquely identify a subproblem.*

So, in the case of Fibonacci, this is simply the parameter n. We should aim to make this set of parameters as small as possible to reduce the number of possible states. The set of states a DP problem can take is called the *state space*. For Fibonacci the state space is the numbers $0, \ldots, n$. For the shortest path problem, the state space is any possible location, i.e., any node in the graph. For the money problem in Appendix **??**, it is all possible combinations of a valid amount, and the largest banknote you are still allowed to use.

Step 2 After identifying the state space, we need to identify state transitions. That is, which states are reachable from which states. This can be directly read from the recurrent relation. For example, if we are in state n for the Fibonacci numbers, the possible next states are $n-2$ and $n-1$. Therefore, if we know the values (i.e., the Fibonacci numbers) for $n-1$ and $n-2$, we can compute the value for state n. Another example: if we are in a node in the graph, the possible next states are its neighbours. Therefore, if we know the shortest paths for all a nodes neighbours, we can compute the shortest path for the node.

Step 3 Identify the states with a known value, i.e., the base cases. For Fibonacci these are 0 and 1 (with values 0 and 1). For The shortest path problem, this is only the `finish` state, which has a value of 0 by definition, and so on.

Step 4 Put the values for all the states in a table (e.g., an array, a dictionary etc.). Fill the values for unknown states with default values (e.g., ∞ for the shortest path problem).

Step 5 Using the state transitions, iteratively improve the values for the states using the transitions from a given state to the possible next states (e.g., we use the values of $n-1$ and $n-2$ to compute the value for state n in Fibonacci). This can be a while-loop, if the number of necessary iterations is unknown, but it can be checked whether we should stop. Or it can be a for-loop, in the case that the number of necessary iterations is known beforehand.[31]

So, in general, a DP algorithm with take $O(maxIter|T|)$ times, where $|T|$ is the number of possible state-transition from any state to any other state, and $maxIter$ is the

[31] Note that we could have used a for-loop from 0 to $|V|-1$ in Algorithm 27, rather than a while-loop. The advantage of the while-loop however is that the algorithm might terminate significantly faster.

maximum number of iterations the DP algorithm requires to terminate. Furthermore, a DP algorithm will typically take up $O(|S|)$ space (where $|S|$ is the size of the state-space) as the values for each possible state are determined and improved upon. Note however, that this is an upper bound, and it might be possible to be more efficient than that.

In the shortest path problem, as we have seen, we actually need the full $O(maxIter|T|) = O(|V|\ |E|)$ runtime. This is because the value estimate for each state may change in each iteration of the algorithm.

However, in many algorithms, we can be more efficient. In Fibonacci for example, we observe that once the value for a state becomes known, it never changes anymore. Hence, we only need to visit every possible state transition once. And because there are only $|T| = 2|S|$ state-transitions, we can do with a time complexity of $O(|S|) = O(n)$. This gives rise to algorithm 28.

Algorithm 28 DP Fibonacci

Input: a non-negative integer n
Output: the n-th Fibonacci number

```
1: fibo ← an array of length n + 1
2: fibo[0] ← 0
3: fibo[1] ← 1
4: for i ∈ 2...n do                    ▷ Every state only needs to be visited once.
5:     fibo[i] ← fibo[i − 1] + fibo[i − 2]
6: end for
7: return fibo[n]
```

The same is true for the Money problem. We can just fill the matrix bottom up.

16. Complexity Classes and Really Hard Problems

In the previous section, we have seen a graph-based problem that was solvable in polynomial time. Specifically, the shortest-path problem, which was solvable in $O(|V|^2)$ time with Dijkstra's algorithm, and $O(|V||E|)$ with the Bellman-Ford algorithm. However, in most graph-based problems, we will not be that lucky. In this section, we will go into the complexity of problems, and introduce complexity classes for computational problems.

The complexity class that you are most familiar with is called P: this is the class (set) of all problems that can be solved in polynomial time in terms of the size of its inputs. We have discussed polynomial runtime in Section 2.3. The shortest-path problem is in the complexity class P because there are algorithms that solve it in polynomial time: the Bellman-Ford algorithm in the number of edges times the number of nodes (this is polynomial), and Dijkstra's algorithm quadratic in the number of nodes (this is polynomial). This immediately shows how you can prove a certain computational algorithm is in P: simply formulate an algorithm, and show that its worst-case time complexity is order $O(n^c)$, where c is a constant. Alternatively, if there are multiple

inputs, whose sizes are n, m and o, then the worst-case time complexity should be $O(n^c m^d o^e)$, where c, d, and e are constants.

As computer scientists, we love P! When a problem is in P, we can be relatively sure how long computation will take, and that given the right hardware and a good programmer and compiler, we can make it work. Therefore, a lot of what the computer science community does is show that problems are in fact in P, by writing algorithms for them that have polynomial time complexity. Then, when this is known, new algorithms may be introduced that have *better* polynomial complexity, i.e., lower exponents c in $O(n^c)$. For example, Dijkstra's algorithm was introduced later than the Bellman-Ford algorithm, and is an improvement because $O(|V||E|)$ is worse than $O(|V|^2)$, as graphs can contain significantly more edges than nodes (at worst $|V|^2$, when every node in the graph connects to every other node). Finally, algorithms can be improved by giving them a lower memory-complexity. Alternatives may also be devised that have a better expected runtime (as may be the case when randomisation is used). This typically offers a palette of algorithms for a problem that offers different trade-offs between worst-case time complexity, memory complexity, and expected runtime.

Sadly for us though, not all computational tasks are in P. Specifically, many real-world problems cannot be solved in polynomial time.

NP-Complete problems are problems for which no polynomial time ($O(n^c)$ for any constant c) exist[32] that solves the problem, but – and this is important – any given *potential solution* to the problem can be checked in polynomial.

NP-hard problems are *at least as hard* as NP-complete problems. So, again no algorithms exists that solve the problem in polynomial time, but it is now also no longer clear whether a potential solution can be checked in polynomial time.

Even harder problems also exist for which it has been proven that solving the problem takes up at least exponential memory and time, or even worse than that.

In practice, we will not care much for even more difficult problems. This is because as soon as a solution is no longer in P, we will need exponential time ($O(c^n)$, where $c > 1$) to solve the problem exactly, and this is simply infeasible for problems that grow large. Therefore, we typically just have to *give up* on finding an optimal solution for these problems, and have to resort to *approximate/heuristic* algorithms instead.

16.1. Examples of NP-complete problems

We cannot solve (reasonably large) NP-complete (or worse) problems in reasonable time. However, they are rather abundant in the real world. A few very common NP-complete problems are listed below. Please note that these are all problems which have a yes-or-no

[32] most likely – and at least not for now. There are still people who hope to find polynomial runtime algorithms for NP-complete problems, however, people have been trying for a very long time.

question attached to them. A solution is an example for the answer "yes". When such a solution is found we can check that it is indeed a solution in polynomial time.

- The Boolean *satisfiability* problem (SAT) – given a Boolean expression, e.g., (*a or b*) *and* (*not a and* (*no*
  is there a value for the variables (a, b, and c) that makes this expression **true**? Once we have a solution, e.g., $a = false; b = true; c = true$, we can check this indeed makes the expression true (or not) in polynomial time[33]. This problem naturally pops up in all kinds of computer science, from requirements engineering to electronic circuits.

- The k graph colouring problem (GC) – can we assign a colour to each node in a graph, with at most k colours, such that no neighbouring nodes (nodes directly connected by an edge) have the same colour? See Figure 23 for an example of this problem and an example solution. The classic example of graph colouring is colouring the countries on a map, such that no two adjecent countries have the same colour.

  The graph colouring problem seems rather abstract, but it does in fact have many key applications in computer science and beyond. For example, when the compiler needs to decide which registers to use for which variables at cannot assign the same register to two variables that need to be in scope at the same time. Therefore, we can express the variables in a program (that need a register) as nodes in a graph, and put an edge between the variables that are in scope at the same time (cannot be assigned to the same register), then if see the number of free purpose registers as our colours, voilà, we have a graph colouring problem. Furthermore, note that the popular game Sudoku is in fact a (very specific of course) graph colouring problem. And, more general, any scheduling problem with a limited number of resources (such as machines in a workshop) and tasks that use such resources can be cast as a graph colouring problem, where the question is which tasks can I do in parallel? If two tasks have a different colour, they can be done in parallel, but not is they have the same colour. This is because two tasks that require usage of the same machine cannot happen at the same time.

- The travelling salesman decision problem (TSP-d) – If we have a road network, with distances on the edges, is there a cycle (i.e., starting and ending in the same node), that visits a list of nodes for which the entire round-trip is less than l? This is of course directly applicable to the logistics sector, in which deliveries must be made to multiple addresses, within the distance e.g., an electric van can travel on one charge (minus a comfortable margin). But it also has applications in manufacturing and electrical wiring.

So, as these problems are in fact very common in all kinds of real-world problems, we want to have good algorithms to solve them. However, as NP-complete problems (and

[33] We omit the proof for this in the reader. Please check out the Cook–Levin theorem if you want to know more.

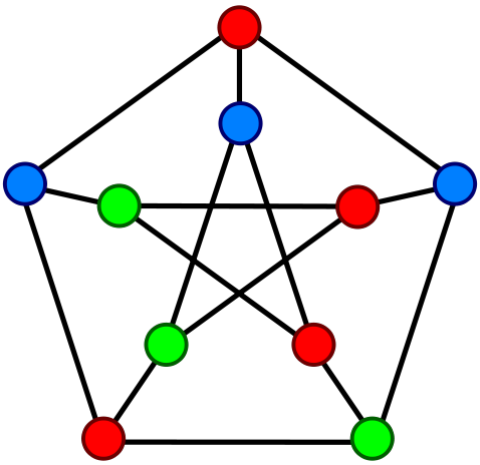

Figure 23: Example solution for the graph colouring problem, with $k = 3$ (3-GC). Figure taken from `https://commons.wikimedia.org/wiki/File:Petersen_graph_3-coloring.svg`, own work by Chris-martin.

by extension NP-hard or worse) problems the only known exact algorithms have exponential time-complexity (or worse). This is of course highly undesirable. If a problem with an input of say 10 integers takes about $1ms$ to compute, an input of 77 integers (i.e., a problem not even 8 times larger), can take about the age of the earth to solve exactly. Therefore, in practice, we typically rely on heuristic algorithms to find approximate solutions to problems that are NP-complete (or harder, so NP-hard or worse).

16.2. Determining NP-completeness and NP-hardness *

This section is for the diehards!

So, how can we know whether a new computational problem is NP-hard or NP-complete? To show NP-hardness the proofs usually make use of the known NP-complete problems, i.e., if we can show that any problem formulated as a known NP-complete problem can be written as an instance of the new problem, the new problem is NP-hard.

A fun example of a NP-hardness is the 2000 proof by Richard Kaye [7] that Minesweeper (the popular game that used to be installed on Windows machines) is in fact NP-hard. To do so, he proved that SAT-problems (defined above) can be written as a Minesweeper problem. That is, SAT problems can be written as electronic circuits with logic gates. These can be recreated in Minesweeper. Therefore any SAT-problem can be written as Minesweeper instance (if the field is large enough), and thus, Minesweeper is NP-hard.

In fact, Kaye also shows that Minesweeper is not harder than the SAT-problem either: Minesweeper can be written as a SAT-problem (of bounded size that is polynomial or better in the size of the original Minesweeper instance). Hence, Minesweeper is NP-complete.

16.3. Even worse?*

This section is for the diehards!

So, this begs the question, are there complexity classes which are even worse? Well, NP-hard indicates a lower bound, so NP-hard already contains significantly more difficult problems. However, there are also problems that can be proven to not be in P, i.e., problems that can provable only be solved in exponential runtime (but not worse). These problems are $EXPTIME$-complete.

Finally, there is a very nasty class of problems: those that are *undecidable*. These are problems for which it is proven that it is impossible to construct an algorithm that always leads to a correct answer. An example of this is the (infinite-horizon) *partially observable Markov decision process* [12], i.e., for example: a robot that needs to navigate a building to pick up trash at night, but does not know the state of the building (i.e., where there is trash, and which locations are inaccessible due to being blocked) when it starts to operate. Good strategies will balance getting to know where there is trash, as well as pick it up, but there is no algorithm that is guaranteed to produce the optimal strategy to do so for all instances of such a problem.

16.4. Optimisation problems

Most optimisation problems (see Section 10) are more difficult than the NP-complete problems mentioned above. This is most easy to see for the Travelling Salesman Problem (TSP). Specifically, there are two well-known versions of this problem:

- The decision version (TSP-d) (discussed in Section 16.1): is there a cycle (a route from A to A) visiting all the locations in a given list with a length of at most l? This problem is NP-complete, and has a yes or no answer.
- The optimisation version (TSP-o): what is the shortest cycle that visits all locations in a given list?

TSP-o is harder than TSP-d. An easy way to see this to see what happens if we present a solution, and ask the question: is this a correct solution. For TSP-d, the question is: does this solution, i.e., a given cycle, indeed have a length less than l. This can be checked in polynomial time. For TSP-o, the question is: is this indeed the cycle with the minimal length? To answer this question, we need to answer the question whether there is a cycle with a smaller length. So, if the length of the presented solution is x, is there a cycle with a length smaller than x. This is a type of question that we know: it is the TSP-d problem.

TSP-o is NP-hard, which is not NP-complete but harder! In fact, there are more complexity classes out there to describe exactly what type of complexity TSP-o, and similar optimisation problems have. However, we will not discuss this any further in this reader because for all practical purposes the implication is the same: *it is infeasible to solve such problems exactly.* We therefore move on to first, introducing a very general optimisation problems (that encompasses many other optimisation problems), called the coordination graph in Section 16.5. Then we will go into algorithms to solve such

extremely hard problems by using heuristic approaches, specifically we will look at *local search (LS)* algorithms and improvements upon those, in Section 17.

16.5. Coordination graphs

A very generally formulated optimisation problem is what is known as a *coordination graph* (CoG). Because this problem is so general, it has been used by many different subfields of computer science, artificial intelligence and mathematics, under a different name. For example, CoGs are also called *weighted constraint satisfaction problems (WCSPs)* and *distributed constraint optimisation problems (DCOPs)*.[34] A CoG is a tuple $(\mathcal{D}, \mathcal{A}, \mathcal{U})$, where

- $\mathcal{D} = \{1, ..., n\}$ is the set of n decision variables (or agents). These are the nodes in the graph.
- $\mathcal{A}$ is the search space[35]. The search space consists of all the possible values for each of the decision variables. Let us write $\mathcal{A}_i$ for the set of values that decision variable i can take. For example, if i corresponds to a robot in robot football, $\mathcal{A}_i$ could be $\{\texttt{move\_forwards}, \texttt{rotate\_left}, \texttt{rotate\_right}\}$. A specific complete solution is thus a tuple containing a value/action for each decision variable/input $\mathbf{a} = (a_1, ..., a_n)$. We will write bold $\mathbf{a}$ for full solutions and normal a_i for the possible values of individual decision variables.
- Finally, $\mathcal{U} = \left\{u^{i,j}, ..., u^{y,z}\right\}$ is the set of ρ *local reward functions*, each of which has limited *scope*, i.e., it depends on only a (small) subset of the decision variables. In this reader we only consider binary interactions, i.e., local reward functions that depend on two decision variables. These local reward functions are the edges in the graph. Each local reward function is a table with rewards for each possible (joint) value of the decision variables that it connects.

The goal in a CoG is to maximise the total (team) reward, which is the sum of all the local rewards, by finding the best solution $\mathbf{a}$. The problem with this is that while local reward depends on only two decision variables, a decision variable may participate in more than one local reward function. For example, consider the following CoG (Figure 24, where the total team reward decomposes into two local reward functions, each with two decision variables in scope:

$$\max_{\mathbf{a}} u(\mathbf{a}) = \max_{\langle a_1, a_2, a_3 \rangle} u^{1,2}(a_1, a_2) + u^{2,3}(a_2, a_3).$$

The local reward functions are defined in Table 3.

[34] In this reader we use the name CoG as this happens to be the name most common in the field where the first author of this reader happens to have done most of his research. Yes, this is of course rather arbitrary, but so would every other choice be.

[35] This is also called the solution space, or input space (or just the set of possible inputs). Furthermore, in the multi-agent literature this is called the joint action space

	$\dot{a}_2$	$\bar{a}_2$
$\dot{a}_1$	3.25	0
$\bar{a}_1$	1.25	3.75

	$\dot{a}_3$	$\bar{a}_3$
$\dot{a}_2$	2.5	1.5
$\bar{a}_2$	0	1

Table 3: The local reward tables for $u^1(a_1, a_2)$ (left) and $u^2(a_2, a_3)$ (right). There are two possible values per decision variable, denoted by a dot ($\dot{a}_1$) and a bar ($\bar{a}_1$).

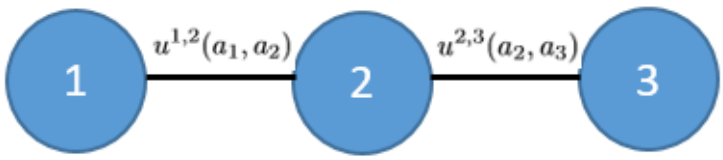

Figure 24: A CoG with 3 agents and 2 local payoff functions.

So if we want to look up the value of a full solution, say $\mathbf{a} = (\bar{a}_1, \dot{a}_2, \bar{a}_3)$, we have to look to the local rewards in each table:

$$u((\bar{a}_1, \dot{a}_2, \bar{a}_3)) = u^{1,2}(\bar{a}_1, \dot{a}_2) + u^{2,3}(\dot{a}_2, \bar{a}_3) = 1.25 + 1.5 = 2.75.$$

The main challenge in a CoG is that the size of the solution space, $|\mathcal{A}|$, grows exponentially with the number of decision variables. Therefore, the naïve brute-force solution strategy: just looping over all possible joint actions, calculating their values, and storing the maximum quickly becomes *intractable* (i.e., not doable in reasonable time). For example, in this three-variable problem with two possible values for each, the number of possible solutions (i.e., the size of the search space) is $2^3 = 8$, but if we have n variables, this would become 2^n.

17. An Introduction to Heuristic Search Algorithms

Many optimisation problems – like the coordination graphs (CoGs) – in the previous section cannot be solved exactly in polynomial time. Because an algorithm with a time complexity of $O(2^n)$ is typically infeasible to use, we therefore tent to use approximate algorithms.

17.1. Local Search

In this section, we introduce a popular class of approximate algorithms called *local search (LS)*. Local search algorithms operate on *local optimisation*, i.e., we input a solution, select 1 (or 2 or 3) decision variables, and see whether we can improve the solution by changing the value for that variable. In other words, we make local improvements to the solution.

For example, in the previous section we saw an example coordination graph in Figure 24 and Table 3. When we start with a solution $\mathbf{a} = (\bar{a}_1, \dot{a}_2, \bar{a}_3)$, and the associated team reward:

$$u((\bar{a}_1, \dot{a}_2, \bar{a}_3)) = u^{1,2}(\bar{a}_1, \dot{a}_2) + u^{2,3}(\dot{a}_2, \bar{a}_3) = 1.25 + 1.5 = 2.75,$$

we can make improvements by taking one of the variables, e.g., lets take number 3, and see wheter we can improve the solution by changing the value of a_3 from $\bar{a}_3$ to a different value. In this case the only alternative we have is $\dot{a}_3$. Making this change has an effect on one local payoff function, i.e., we would move from $u^{2,3}(\dot{a}_2, \bar{a}_3) = 1.5$ to $u^{2,3}(\dot{a}_2, \dot{a}_3) = 2.5$. So this indeed improves the value of the total solution. The local effect is $\Delta = u^{2,3}(\dot{a}_2, \dot{a}_3) - u^{2,3}(\dot{a}_2, \bar{a}_3) = +1$, so the value improves to $u((\bar{a}_1, \dot{a}_2, \dot{a}_3)) = u((\bar{a}_1, \dot{a}_2, \bar{a}_3)) + \Delta = 3.75$[36] This is an illustration of why local changes can be evaluated quickly:

- Local changes only have a local effect (i.e., change the reward for a small number of local reward functions).
- So we should calculate the local change, Δ.
- And then we add this local change to the total team reward to obtain the new team reward.

This brings us to the algorithmic scheme of local search, in Algorithm 29. Note that this algorithm terminates when no local improvements are possible anymore. In this case, the solution is a so-called *local optimum*. A local optimum is of course much better than a random solution. but it is not necessarily the best solution for the problem. In fact, it usually is not.

[36]Please note that this value is equal to what we would get if we would do the calculation of the team reward for the new solution from scratch: $u((\bar{a}_1, \dot{a}_2, \dot{a}_3)) = u^{1,2}(\bar{a}_1, \dot{a}_2) + u^{2,3}(\dot{a}_2, \dot{a}_3) = 1.25 + 2.5 = 3.75$.

Algorithm 29 Local Search (LS)

Input: a difficult optimisation problem, `problem`,
and a (typically random) start solution **a**.
Output: an approximate solution **a** (a local optimimum)

1: $vars \leftarrow$ a queue of the decision variables in random order
2: **while** not $vars.empty()$ **do**
3: $i \leftarrow vars.dequeue()$
4: **for** all possible values a_i of decision variable i **do**
5: $\Delta \leftarrow$ `problem.evaluateChange`$(\mathbf{a}, i, a_i)$
6: **if** $\Delta > 0$ **then**
7: Change **a** by changing the value of variable i to a_i
8: $vars \leftarrow$ a queue of the decision variables in random order ▷ After an improvement for one variable, new improvements may be possible for the other decision variables again.
9: break for-loop ▷ Continue at while-loop with the new queue $vars$
10: **end if**
11: **end for**
12: **end while** ▷ When vars is empty, there are no more possible local improvements
13: **return a**

17.2. Multi-start Local Search

Local search terminates when a local optimum, **a**, is found. However, local optima are typically not actual optima. (Of course, the global optimum is also a local optimum though, just not the other way around.) To avoid getting stuck in local optima, there are different strategies. The simplest is multi-start local search. That is, we just run local search for n times, and keep the best.

Algorithm 30 Multi-start Local Search (MLS)

Input: a difficult optimisation problem, `problem`, and a number of trials, n
Output: an approximate solution $\mathbf{a}_{mls}$ (a local optimum), and its value

1: $\mathbf{a}_{mls} \leftarrow$ null
2: $val \leftarrow -\infty$ ▷ negative infinity
3: **for** $i \in 0 \ldots n-1$ **do**
4: **a** $\leftarrow$ a random solution ▷ a random value for each decision variable
5: **a** $\leftarrow LS($`problem`$, \mathbf{a})$ ▷ Algorithm 29
6: $newVal \leftarrow$ `problem.evaluate`$(\mathbf{a})$
7: **if** $newVal > val$ **then**
8: $\mathbf{a}_{mls} \leftarrow \mathbf{a}$
9: $val \leftarrow newVal$
10: **end if**
11: **end for**
12: **return** $\mathbf{a}_{mls}$ and val

17.3. Iterated Local Search

Multi-start local search (MLS) is a very straight-forward approach, and easy to code. However, it starts from scratch every iteration. iterated local search (ILS) on the other had, does not start from scratch, but aims to shake things up enough to escape a local optimum, yet still maintain enough of the local optimum to exploit its good parts. To achieve this, it uses a (small) perturbation probability p. After each local search, it takes the local optimum, $\mathbf{a}$, and then for each decision variable changes the value with probability p. Note that if $p = 1$, iterated local search becomes multi-start local search.

Algorithm 31 Iterated Local Search (ILS)

Input: a difficult optimisation problem, `problem`, a number of trials, t and a perturbation probability p.
Output: an approximate solution $\mathbf{a}_{ils}$ (a local optimum), and its value

1: $\mathbf{a}_{ils} \leftarrow$ a random solution ▷ an array of length n (the number of decision variables), with a random value for each decision variable
2: $val \leftarrow -\infty$ ▷ negative infinity
3: **for** $i \in 0 \dots t-1$ **do**
4: $\quad \mathbf{a} \leftarrow \mathbf{a}_{ils}$
5: $\quad$ **for** each decision variable, j **do**
6: $\quad\quad r \leftarrow$ draw a random number between 0 and 1
7: $\quad\quad$ **if** $r < p$ **then**
8: $\quad\quad\quad$ Change $\mathbf{a}$ by changing the value of variable j to a random value
9: $\quad\quad$ **end if**
10: $\quad$ **end for**
11: $\quad \mathbf{a} \leftarrow LS(\texttt{problem}, \mathbf{a})$ ▷ Algorithm 29
12: $\quad newVal \leftarrow \texttt{problem.evaluate}(\mathbf{a})$
13: $\quad$ **if** $newVal > val$ **then**
14: $\quad\quad \mathbf{a}_{ils} \leftarrow \mathbf{a}$
15: $\quad\quad val \leftarrow newVal$
16: $\quad$ **end if**
17: **end for**
18: **return** $\mathbf{a}_{ils}$ and val

17.4. Genetic/Evolutionary Algorithms

Local search algorithms use local mutations to find local optima. To escape local optima, we have seen that we can run local search many times with different initialisations (MLS) or perturb and continue (ILS).

LS, MLS and ILS only try to optimise one solution at a time. However, if we are generating multiple solutions anyway, can we not combine the good parts of multiple solutions to come up with a better solution? This is the central idea behind evolutionary/genetic algorithms.

Evolutionary (/genetic) algorithms are inspired by the process of biological evolution.

Basically, we view a solution $\mathbf{a}$, which is an array of values for the different decision variables, as the genome. The reward corresponding to this solution $u(\mathbf{a})$ is called the *fitness* in an evolutionary context; fitter individuals have a higher chance of producing offspring (and sometimes to survive longer themselves). In other words, at every iteration an evolutionary algorithm:

- Starts with a *population* of (i.e., a container filled with) solutions. The population size N is a parameter of the algorithm. This population can be initialised with random solutions, or e.g., MLS.
- From the population the algorithm selects pairs of individuals to "reproduce". This is done using a *cross-over* operator. A cross-over operator is a function that takes two solutions $\mathbf{a}$ and $\mathbf{b}$ (the parents), and produces a new solution $\mathbf{c}$ (offspring) that combines elements of both $\mathbf{a}$ and $\mathbf{b}$.
  - How to select solutions for reproduction is a central question in evolutionary algorithm design. Many schemes for doing this exist. It is important that solutions with a high fitness (reward) have a higher probability of reproducing. However, other factors may also contribute (e.g., we may not want to recombine solutions that are too similar).
  - Which cross-over operator to use is the other central question in evolutionary algorithm design. Which operators are good depend on the type of problem to solve, i.e., a cross-over operator for the travelling salesman problem will look very different to a cross-over operator for graph colouring.
- After enough offspring is generated to fill the next generation (i.e., to fill the population size, N), all these new solutions are evaluated, and the process repeats.

The selection function, and the cross-over operator are functions that are given to an evolutionary/genetic algorithm as parameters. (Simple) examples for these functions are:

- `select2(population)`: randomly select two solutions from the top half (in terms of fitness/reward) of *population*.
- `crossOver`$(\mathbf{a}, \mathbf{b})$: copy half of the solution from $\mathbf{a}$ and the rest from $\mathbf{b}$. For example, if solutions are represented as a Python list
  return $\mathbf{a}$[: `len(a)/2`] + $\mathbf{b}$[`len(a)/2` :].

These are of course very simplistic examples. Especially this cross-over operator will not make much sense for most problems, as a range of indices in the list representing the solution does typically not represent a useful way to divide up the problem into parts. Much more sophisticated instances of these operators exist in the literature.

17.5. Genetic Local Search

Genetic/evolutionary algorithms have been developed outside of the context of local search algorithms. However, it is very well possible to combine the two. That is, when,

in a genetic algorithm a new solution is created using cross-over, we can run local search on it. Furthermore, it is also possible to add the iterative scheme again: first do cross-over on two solutions to get a new solution $\mathbf{a}$, then perturb $\mathbf{a}$, and then run local search on $\mathbf{a}$. This leads to Genetic Local Search (Algorithm 32). Genetic local search is one of the most powerful schemes for heuristic optimisation.

Algorithm 32 (A possible scheme for) Genetic Local Search (GLS)

Input: a difficult optimisation problem, `problem`, a number of generations, G, a population size N, a *crossOver* function, a *select*2 function to select two parents from a population of solutions, and a perturbation probability p.

Output: an approximate solution $\mathbf{a}_{gls}$ (a local optimum), and its value

$population \leftarrow$ an empty container of tuples ▷ (solution, reward)
for $i \in 0 \dots N-1$ **do**
    $\mathbf{a} \leftarrow$ a random solution
    $\mathbf{a} \leftarrow$ **localSearch**($\mathbf{a}$) ▷ Algorithm 29.
    $population.add(\ \mathbf{a}, problem.evaluate(\mathbf{a}))\ )$
end for
$\mathbf{a}_{gls}, reward \leftarrow$ the best solution in $population$
for $i \in 0 \dots G-1$ **do**
    $newPop \leftarrow$ an empty container of tuples ▷ (solution, reward)
    for $i \in 0 \dots N-1$ **do**
        $\mathbf{a}_1, \mathbf{a}_2 \leftarrow select2(population)$ ▷ select two parents
        $\mathbf{a} \leftarrow crossOver(\mathbf{a}_1, \mathbf{a}_2)$
        for each decision variable, i **do**
            $r \leftarrow$ draw a random number between 0 and 1
            if $r < p$ **then**
                Change $\mathbf{a}$ by randomly changing the value of variable i
            end if
        end for
        $\mathbf{a} \leftarrow localSearch(\mathbf{a})$ ▷ Algorithm 29.
        $newPop.add(\ \mathbf{a}, problem.evaluate(\mathbf{a}))\ )$
    end for
    $population \leftarrow newPop$
    $\mathbf{a}_{gls}, reward \leftarrow$ replace $\mathbf{a}_{gls}$ by the best solution in $population$ if (and only if) it is indeed better
end for
return the best solution $\mathbf{a}_{gls}, reward$

17.6. A Critical Note on Heuristic Optimisation Algorithms

Heuristic algorithms are key to generating useful approximate solutions to many real-world problems within reasonable time. Because of this, the literature on heuristic algorithms is vast, and there are many scientific conferences dedicated to just (one type

of) heuristic algorithm(s).

Please be ware though: it is easy to go overboard with heuristic algorithm design, and get lost in ever more bells and whistles to add on. Every now and then – even in the scientific field – a paper pops up that casts severe doubt on the usefulness of a complex but often popular algorithm for problems they have in fact been applied to.[37]

Because of this we strongly recommend a careful, borderline minimalistic approach to heuristic algorithm design, i.e.,

- Simple things first: always start with local search.
- Analyse the structure of the problem. This will help you make simple but key improvements. For example, in graph colouring it might be worth trying to swap the values of two neighbouring decision variables in local search (rather than only changing the value of one decision variable). Such a slightly different local search algorithm might work wonders on problems with different structures.
- After local search, only make small add-ons. Baby steps does it. Start with multi-start local search, before implementing iterated local search. If iterated local search does not lead to better results, ask yourself why? Maybe the perturbation probability is too high. Or maybe seemingly small changes irreparably destroy the quality of solutions to the problem (this can happen often in TSP). Or maybe the perturbation probability is too low and you end up getting back to the same local optima.
- Only once you think you have perfected the purely local-search based strategies, start adding genetic ideas. Genetic operators are difficult. Sometimes they do not contribute much. Always compare to your simpler baseline algorithms (your perfected versions of MLS and ILS).
- Test every add-on thoroughly and keep experimenting. While you sleep, your computer can tell you (by running extensive tests for a couple of hours) how useful your add-ons of the previous day have actually been.
- And finally do an *ablative* study. That is: even though every add-on may have contributed to the performance of your algorithm at the time you added it, one you have say 12 add-ons, not all of them might be contributing *now*. Every now and then, try removing each component, in order to test whether it is truly helpful. Components that do not contribute to the performance of your algorithm do not spark joy, and should be tossed out.

18. In Conclusion

In this reader, and during this course, you have studied the analysis and design of algorithms. You now know how to determine the time (and memory) complexity of

[37] For an illustrative example of which see [17].

an algorithm, and therefore decide which algorithm is better. You have designed and implemented algorithms for various problems, including some of the hardest problems known to computer science. It is our hope that the tools you have acquired in this course will serve you well in your endeavours.

Acknowledgements

We thank Marius Versteegen, Jasper van der Post, Duur Alblas, Mike Hoogendoorn, Maaike Hovenkamp, Nathan Houwaart, Oscar Kromhout, and Matthijs Vos, Leo Jenneskens, and Robert Bezem for their remarks and feedback.

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for organization of information. In *Doklady Akademii Nauk*, volume 146, pages 263–266. Russian Academy of Sciences, 1962.

[2] D. L. Chao, M. E. Halloran, V. J. Obenchain, and I. M. Longini Jr. Flute, a publicly available stochastic influenza epidemic simulation model. *PLoS computational biology*, 6(1):e1000656, 2010.

[3] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[4] M. de Waard, D. M. Roijers, and S. C. Bakkes. Monte Carlo tree search with options for general video game playing. In *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2016.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[6] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[7] R. Kaye. Minesweeper is np-complete. *Mathematical Intelligencer*, 22(2):9–15, 2000.

[8] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[9] J. R. Kok, M. T. Spaan, N. Vlassis, et al. Multi-robot decision making using coordination graphs. In *Proceedings of the 11th International Conference on Advanced Robotics, ICAR*, volume 3, pages 1124–1129, 2003.

[10] G. Koole. *An Introduction to Business Analytics*. Lulu. com, 2019.

[11] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th international symposium on symbolic and algebraic computation*, pages 296–303. ACM, 2014.

[12] O. Madani, S. Hanks, and A. Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems. In *AAAI/IAAI*, pages 541–548, 1999.

[13] M. Newman and G. Barkema. *Monte Carlo methods in statistical physics*. Oxford University Press: New York, USA, 1999.

[14] E. Peer, V. Menkovski, Y. Zhang, and W.-J. Lee. Shunting trains with deep reinforcement learning. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3063–3068. IEEE, 2018.

[15] R. Piscitelli et al. *Pruning techniques for multi-objective system-level design space exploration*. PhD thesis, Universiteit van Amsterdam, 2014.

[16] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[17] T. van den Berg and S. Whiteson. Critical factors in the performance of HyperNEAT. In *GECCO 2013: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 759–766, July 2013.

[18] S. Zhang, S. Bakkers, D. M. Roijers, and P. Spronck. Avatars of a feather flock together: Gender homophily in online video games revealed via exponential random graph (ERG) modeling. *IEEE Transactions on Games*, 2019. To Appear.

A. Assignments

A.1. Writing a Gomoku Agent with Monte-Carlo Tree Search

In the coming two weeks, the purpose is to write an automatic player (an agent) for the game *gomoku*. The rules for this game can be found on Canvas. In short:

- The purpose of the game is to get *exactly* 5 in-a-row. (So, not 6 or more.)
- The game is played on a 19x19 board (like *go*).
- Black moves first.

The agents submitted will be entered into a competition. We (teachers and TAs) will run this competition.

The assignment is split in a *must-have* and a *should/could-have* part. The must-haves need to be implemented correctly in order to obtain a passing grade. So if the must-haves are not correct, you cannot pass the assignment (regardless of the other things you have implemented). Conversely, if the must-haves are implemented correctly, you have a passing grade (5.5). For additional points, you can implement should/could-haves.

Please describe exactly, in a README file (a text file or `.pdf` file) which should/could-haves you have implemented, and where applicable, answer the questions associated with the item.

A.1.1. Must-haves: a working playing agent

The central method for the agent to implement is `move`. This method will get the current state of the board, the last move of the opponent, and the time it has to decide upon a move as arguments. It is key that the agent responds within this time budget. Making an invalid move, or exceeding the time budget, will lead to disqualification from the competition.

The agent needs to adhere to the following requirements:

- It needs adhere to the schema of the MCTS algorithm (Algorithm 21)
- It needs to have a name that includes your name (implemented in the id-method).
- For each implemented function, a runtime complexity analysis (i.e., the big-Oh of the function, plus an explanation for why this is so), needs to be included in the comments above the function.

See `random_agent.py` (inside `code_gomoku_2021.zip` in the description of Practicum 1) on Canvas for the methods that the agent needs to implement.

The code used to run the competition, that implements the game dynamics, and a random strategy, can be found on Canvas. We will have some (small) prizes for the winners (TBA).

A.1.2. Should/Could-haves: a better agent

However, some deviations from the vanilla MCTS algorithm (that is, the basic MCTS version that is described in Section 12) are not only allowed but *encouraged*. Specifically, the exploration strategy (Equation 1) and the roll-out strategy (Algorithm 23), may be changed at will for extra points.

An overview of additional features leading to extra points that we came up with is:

max 1pt The tree in the basic implementation can be started anew every time the agent needs to move. This is of course rather inefficient: we probably have explored the moves that actually happen already. Make sure that you reuse the tree for the previous move (but of course, do not back-propagate values further back than the current state of the board). Please describe how you achieve this, and why it is efficient in your readme file.

max 1pt An opening book. Many strong AI players of well-known games have an opening book, that helps them select moves in the opening phase of the game. For chess for example, there are opening sequences of moves that human players do all the time, and the best known AI players just reuse those. Please explain how you constructed the opening book, and why you believe this is better than just exploring a lot at the start.

max 4.5pts Experiment with different exploration mechanisms, and decide which exploration strategy to use for your submission on the basis of these experiments. Decide how to determine that one exploration method is better than another, and explain how you got to your final best exploration strategy that you submitted. (For tips: you could consider a heuristic order in which moves to explore first, and then there are different values for the constant in Equation 1, or use a different formula [such as crazy-stone[38]].)

max 4.5pts When you get to a node to expand, MCTS uses a simlution or *roll-out policy* until one of the players wins the game. Until now, we have been using a completely random roll-out policy (just placing stones randomly on the board). However, this is of course a poor approximation of how humans, or even artificial agents will actually play. Experiment with different (more intelligent than completely random) roll-out policies, and decide which roll-out policy to use for your submission on the basis of these experiments. Decide how to determine that one type of roll-out policy is better than another, and explain how you got to your final best roll-out policy that you submitted. You can also experiment with doing more than one roll-out per expansion.

Please note that the last two items on the list may individually lead you to a full score for the assignment. However, this will only be so if we grade your work on these items as "excellent". Therefore, we strongly encourage trying something for both. Also, please

[38] `https://hal.inria.fr/file/index/docid/116992/filename/CG2006.pdf`

note that the *maximum* number of points for each item is indicated. This means that you do not automatically get this maximum and you can get partial points.

If you want to make other adaptations to the algorithm, please contact your klassendocent via e-mail or Teams to ask whether this is allowed before doing so. Please document any deviations from the vanilla algorithms well (in comments at the top of the methods where they are implemented).